DATA MINING
LECTURE 4

Frequent Itemsets, Association Rules

Evaluation

Alternative Algorithms

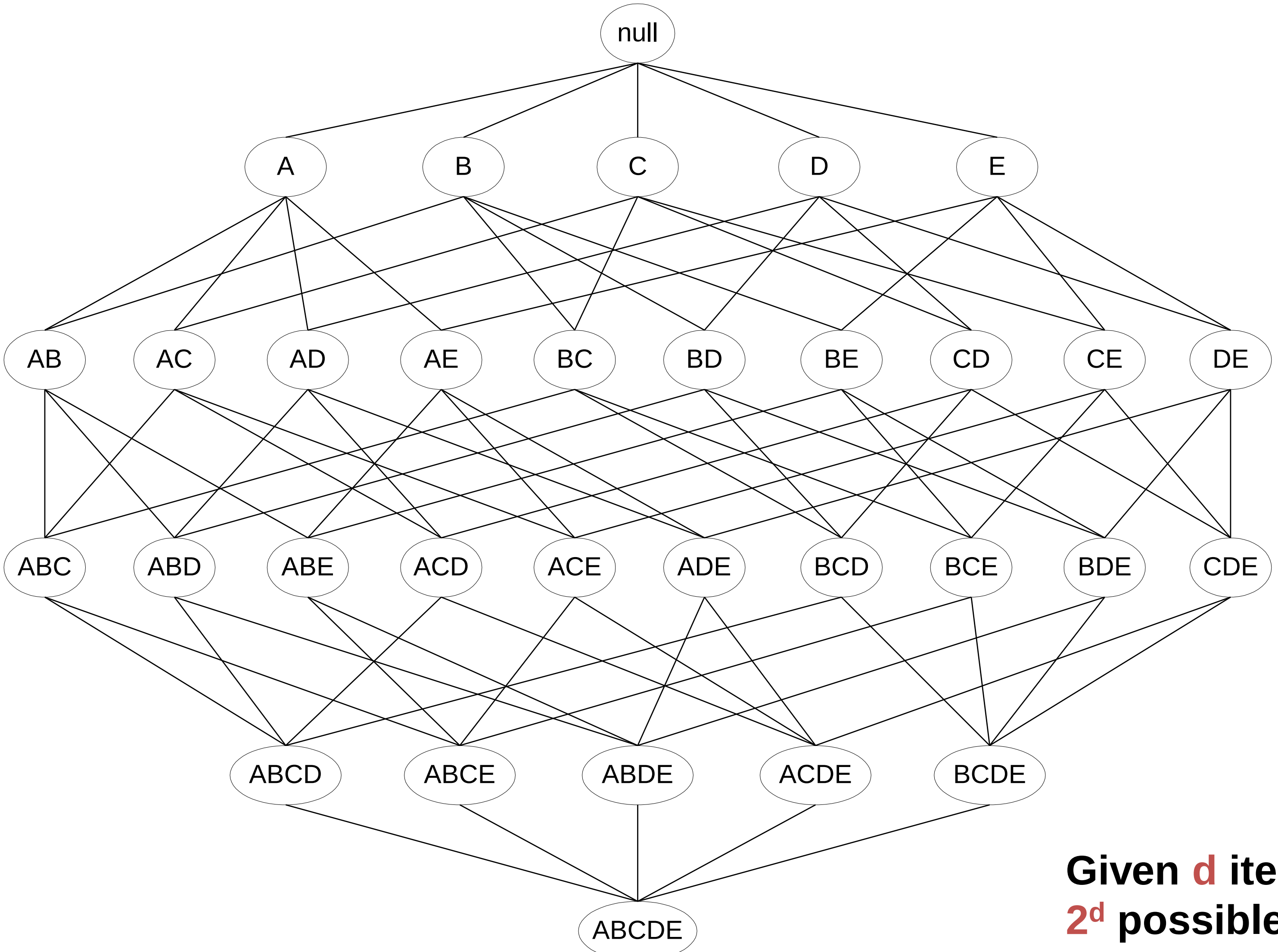
RECAP

Mining Frequent Itemsets

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

- **Itemset**
  - A collection of one or more items
    - Example: {Milk, Bread, Diaper}
  - k-itemset
    - An itemset that contains k items
- **Support (σ)**
  - **Count**: Frequency of occurrence of an itemset
  - E.g. σ({Milk, Bread,Diaper}) = 2
  - **Fraction**: Fraction of transactions that contain an itemset
  - E.g. s({Milk, Bread, Diaper}) = 40%
- **Frequent Itemset**
  - An itemset whose support is greater than or equal to a minsup threshold, $s(I) \geq$ minsup
- **Problem Definition**
  - Input: A set of transactions T, over a set of items I, minsup value
  - Output: All itemsets with items in I having $s(I) \geq$ minsup

The itemset lattice



Given d items, there are 2^d possible itemsets
Too expensive to test all!

The Apriori Principle

- Apriori principle (Main observation):
  - If an itemset is frequent, then all of its subsets must also be frequent
  - If an itemset is not frequent, then all of its supersets cannot be frequent

$$\forall X, Y : (X \subseteq Y) \Rightarrow s(X) \geq s(Y)$$

  - The support of an itemset never exceeds the support of its subsets
  - This is known as the anti-monotone property of support

Illustration of the Apriori principle

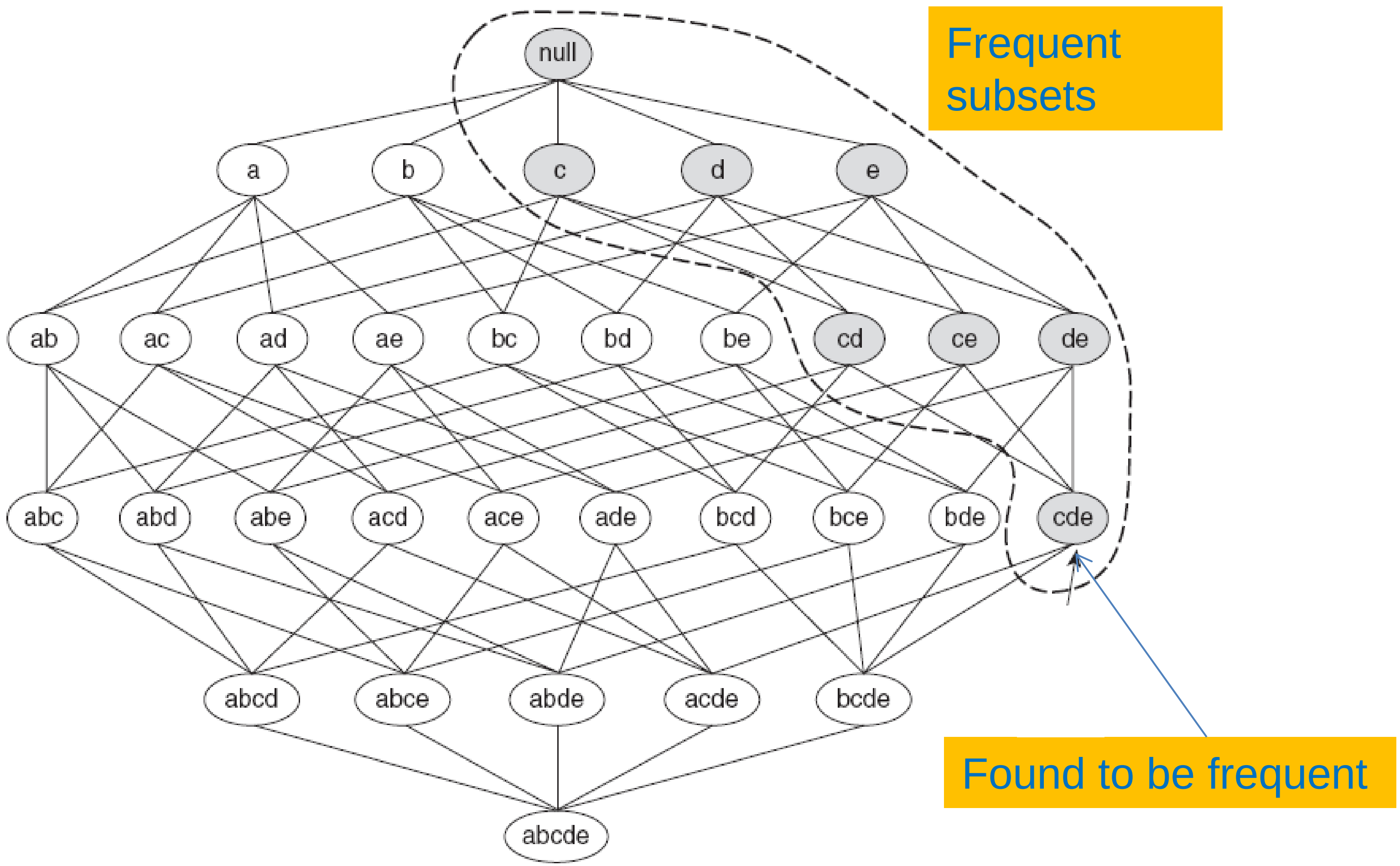
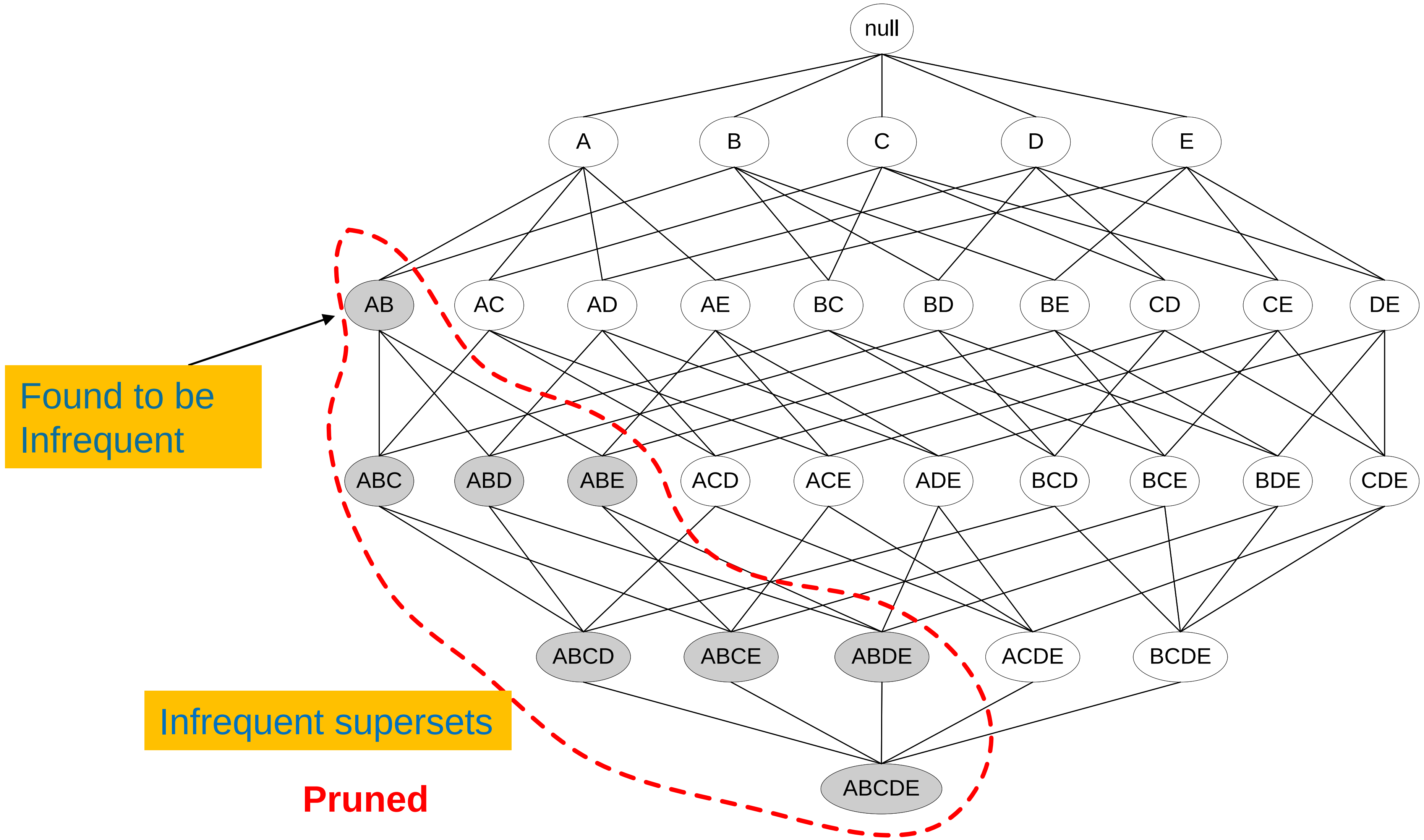


Figure 6.3. An illustration of the *Apriori* principle. If $\{c, d, e\}$ is frequent, then all subsets of this itemset are frequent.

Illustration of the Apriori principle



The Apriori algorithm

Level-wise approach

C_k = candidate itemsets of size k
L_k = frequent itemsets of size k

1. $k = 1$, C_1 = all items
2. While C_k not empty

Frequent itemset generation

3. Scan the database to find which itemsets in C_k are frequent and put them into L_k

Candidate generation

4. Use L_k to generate a collection of candidate itemsets C_{k+1} of size $k+1$
5. $k = k+1$

R. Agrawal, R. Srikant: "Fast Algorithms for Mining Association Rules", *Proc. of the 20th Int'l Conference on Very Large Databases*, 1994.

Candidate Generation

- Basic principle (Apriori):
  - An itemset of size k+1 is candidate to be frequent only if all of its subsets of size k are known to be frequent
- Main idea:
  - Construct a candidate of size k+1 by combining two frequent itemsets of size k
  - Prune the generated k+1-itemsets that do not have all k-subsets to be frequent

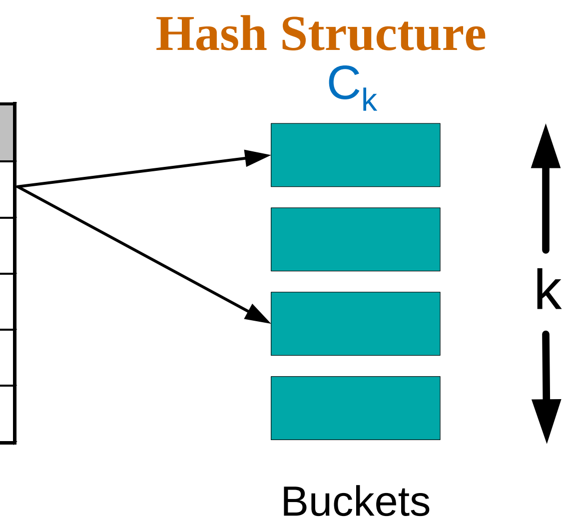
Computing Frequent Itemsets

- Given the set of candidate itemsets C_k, we need to compute the support and find the frequent itemsets L_k.
- Scan the data, and use a hash structure to keep a counter for each candidate itemset that appears in the data

Transactions

N

TID	*Items*
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke



A simple hash structure

- Create a dictionary (hash table) that stores the candidate itemsets as keys, and the number of appearances as the value.
  - Initialize with zero
- Increment the counter for each itemset that you see in the data

Example

Suppose you have 15 candidate itemsets of length 3:

{1 4 5}, {1 2 4}, {4 5 7}, {1 2 5}, {4 5 8},

{1 5 9}, {1 3 6}, {2 3 4}, {5 6 7}, {3 4 5},

{3 5 6}, {3 5 7}, {6 8 9}, {3 6 7}, {3 6 8}

Hash table stores the counts of the candidate itemsets as they have been computed so far

Key	Value
{3 6 7}	0
{3 4 5}	1
{1 3 6}	3
{1 4 5}	5
{2 3 4}	2
{1 5 9}	1
{3 6 8}	0
{4 5 7}	2
{6 8 9}	0
{5 6 7}	3
{1 2 4}	8
{3 5 7}	1
{1 2 5}	0
{3 5 6}	1
{4 5 8}	0

Example

Tuple {1,2,3,5,6} generates the following itemsets of length 3:

{1 2 3}, {1 2 5}, {1 2 6}, {1 3 5}, {1 3 6},

{1 5 6}, {2 3 5}, {2 3 6}, {3 5 6},

Increment the counters for the itemsets in the dictionary

Key	Value
{3 6 7}	0
{3 4 5}	1
{1 3 6}	3
{1 4 5}	5
{2 3 4}	2
{1 5 9}	1
{3 6 8}	0
{4 5 7}	2
{6 8 9}	0
{5 6 7}	3
{1 2 4}	8
{3 5 7}	1
{1 2 5}	0
{3 5 6}	1
{4 5 8}	0

Example

Tuple {1,2,3,5,6} generates the following itemsets of length 3:

{1 2 3}, {1 2 5}, {1 2 6}, {1 3 5}, {1 3 6},

{1 5 6}, {2 3 5}, {2 3 6}, {3 5 6},

Increment the counters for the itemsets in the dictionary

Key	Value
{3 6 7}	0
{3 4 5}	1
{1 3 6}	4
{1 4 5}	5
{2 3 4}	2
{1 5 9}	1
{3 6 8}	0
{4 5 7}	2
{6 8 9}	0
{5 6 7}	3
{1 2 4}	8
{3 5 7}	1
{1 2 5}	1
{3 5 6}	2
{4 5 8}	0

Mining Association Rules

- **Association Rule**
  - An implication expression of the form $X \rightarrow Y$, where X and Y are itemsets
    - {Milk, Diaper} → {Beer}
- **Rule Evaluation Metrics**
  - Support (s)
    - Fraction of transactions that contain both X and Y = the probability P(X,Y) that X and Y occur together
  - Confidence (c)
    - How often Y appears in transactions that contain X = the conditional probability P(Y|X) that Y occurs given that X has occurred.

TID	*Items*
1	**Bread, Milk**
2	**Bread, Diaper, Beer, Eggs**
3	**Milk, Diaper, Beer, Coke**
4	**Bread, Milk, Diaper, Beer**
5	**Bread, Milk, Diaper, Coke**

Example:

$$\{\text{Milk}, \text{Diaper}\} \Rightarrow \text{Beer}$$

$$s = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{|\text{T}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{\sigma(\text{Milk}, \text{Diaper})} = \frac{2}{3} = 0.67$$

- **Problem Definition**
  - Input A set of transactions T, over a set of items I, minsup, minconf values
  - Output: All rules with items in I having s ≥ minsup and c≥ minconf

Mining Association Rules

- Two-step approach:
  1. Frequent Itemset Generation
     – Generate all itemsets whose support $\geq$ minsup

  2. Rule Generation
     – Generate high confidence rules from each frequent itemset, where each rule is a partitioning of a frequent itemset into Left-Hand-Side (LHS) and Right-Hand-Side (RHS)

Frequent itemset: {A,B,C,D}
Rule: AB→CD

Association Rule anti-monotonicity

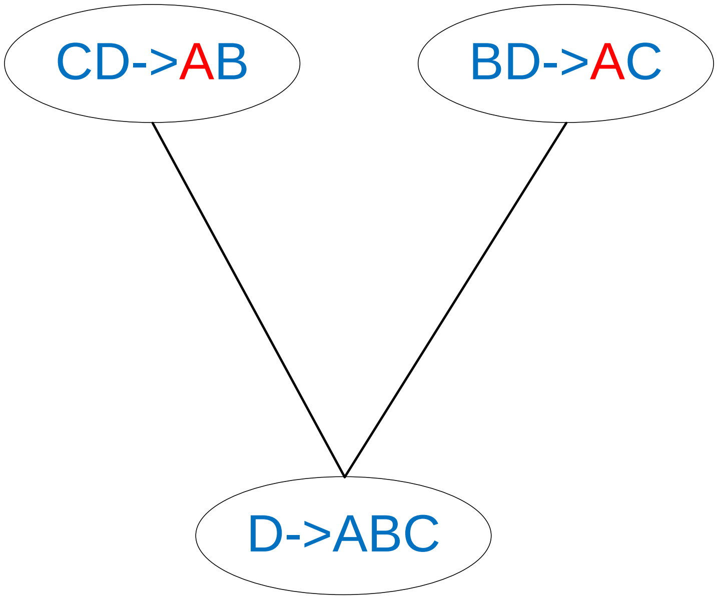
- Confidence is anti-monotone w.r.t. number of items on the RHS of the rule (or monotone with respect to the LHS of the rule)

- e.g., L = {A,B,C,D}:

$$c(ABC \rightarrow D) \geq c(AB \rightarrow CD) \geq c(A \rightarrow BCD)$$

Rule Generation for APriori Algorithm

- Candidate rule is generated by merging two rules that share the same prefix in the RHS



- join(CD→AB,BD→AC) would produce the candidate rule D → ABC
- Prune rule D → ABC if its subset AD→BC does not have high confidence
- Essentially we are doing APriori on the RHS

RESULT POST-PROCESSING

Compact Representation of Frequent Itemsets

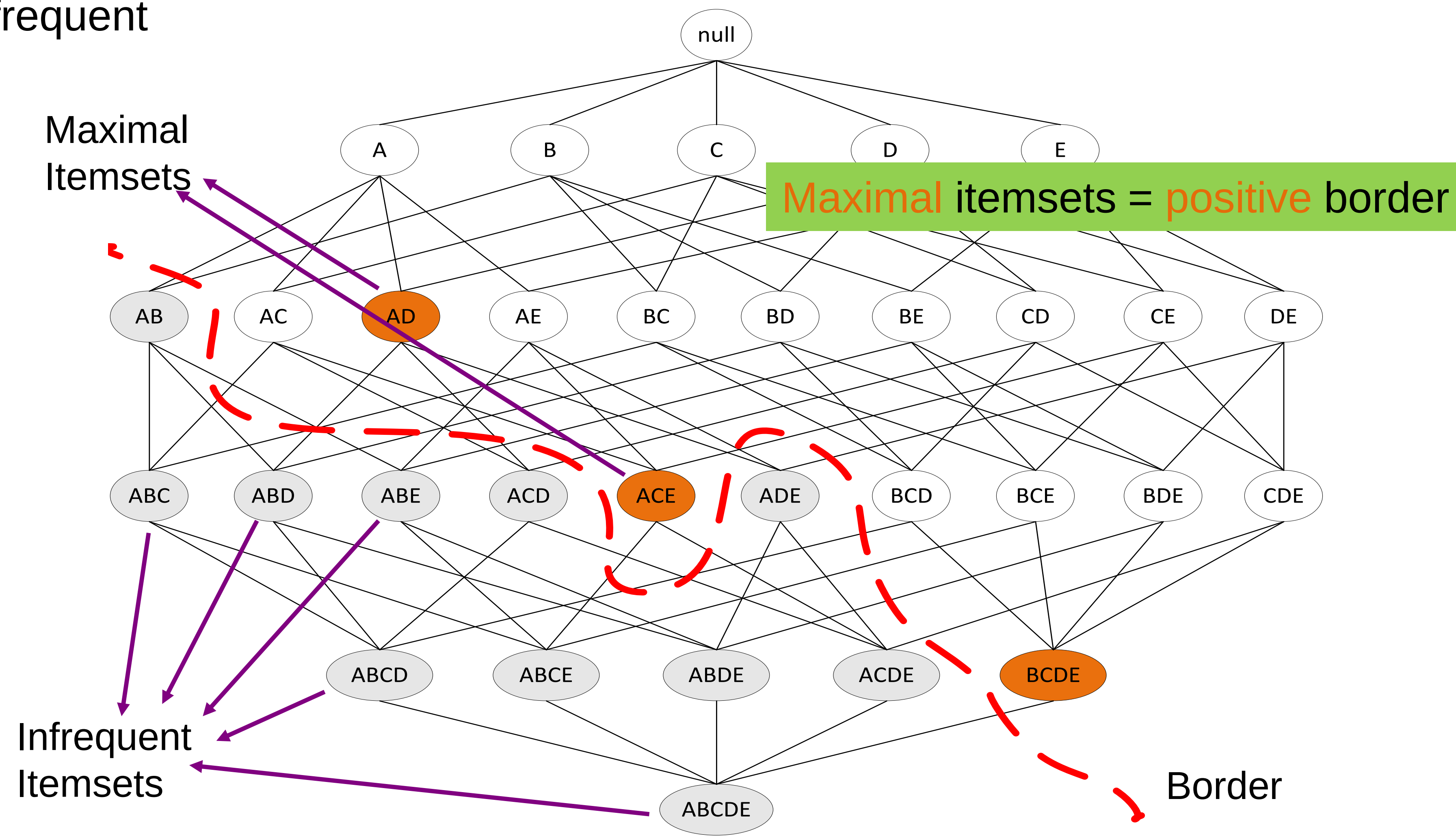
- Some itemsets are redundant because they have identical support as their supersets

TID	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

- Number of frequent itemsets $= 3 \times \sum_{k=1}^{10} \binom{10}{k}$
- Need a compact representation

Maximal Frequent Itemsets

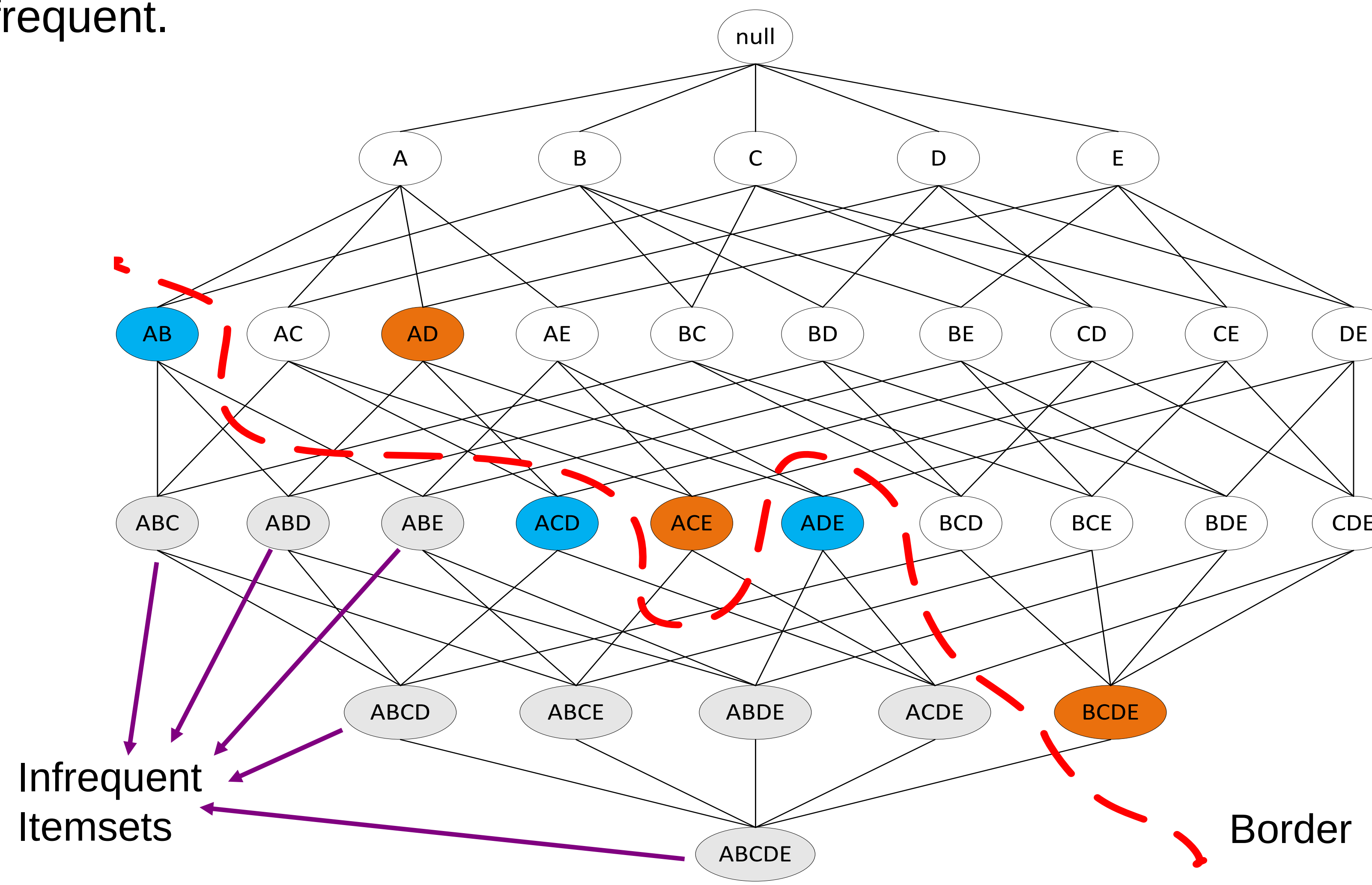
An itemset is maximal frequent if none of its immediate supersets is frequent



Maximal: no superset has this property

Negative Border

Itemsets that are not frequent, but all their immediate subsets are frequent.



Minimal: no subset has this property

Border

- Border = Positive Border + Negative Border
  - Itemsets such that all their immediate subsets are frequent and all their immediate supersets are infrequent.

- Either the positive, or the negative border is sufficient to summarize all frequent itemsets.

Closed Itemsets

- An itemset is closed if none of its immediate supersets has the same support as the itemset

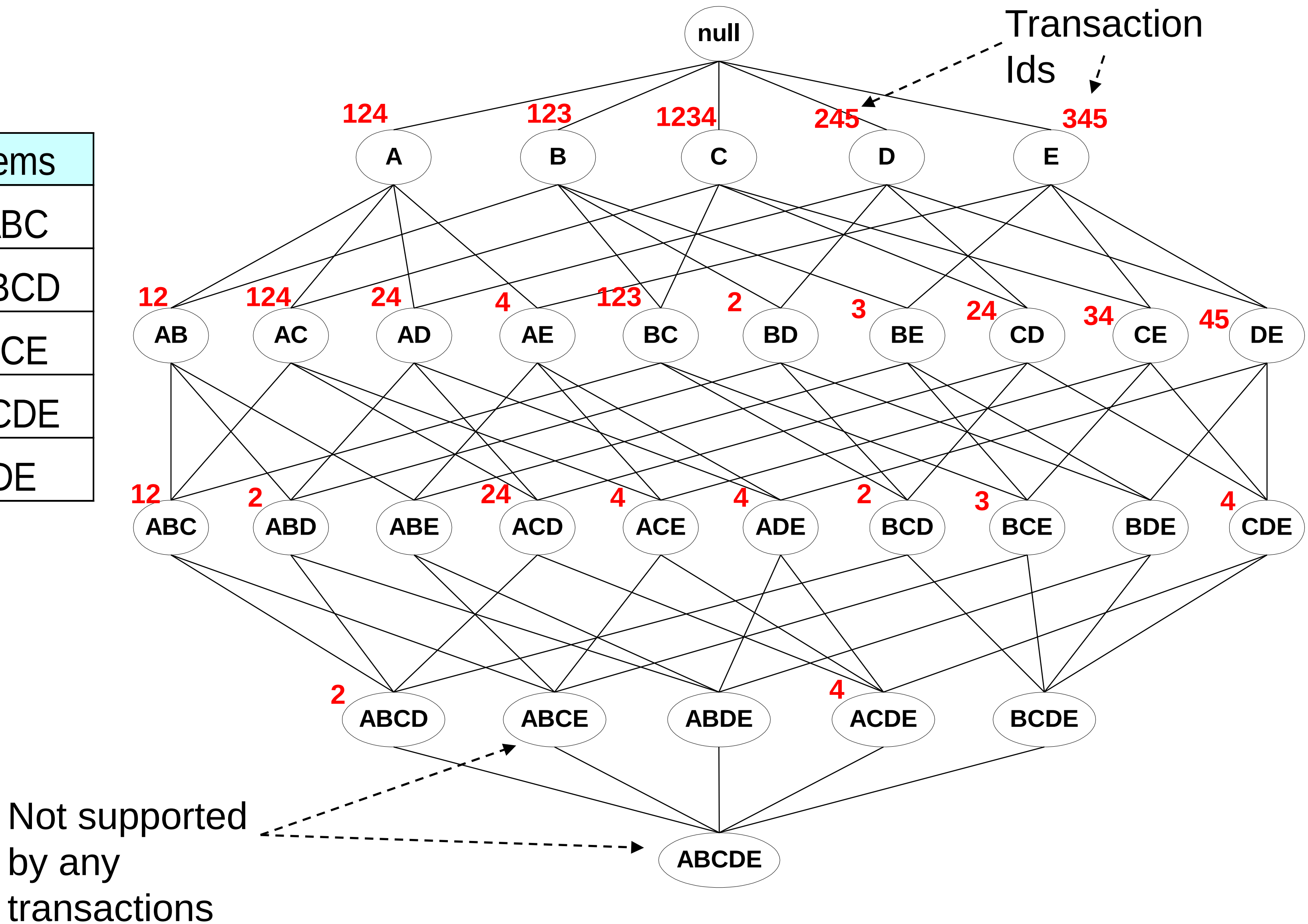
TID	Items
1	{A,B}
2	{B,C,D}
3	{A,B,C,D}
4	{A,B,D}
5	{A,B,C,D}

Itemset	Support
{A}	4
{B}	5
{C}	3
{D}	4
{A,B}	4
{A,C}	2
{A,D}	3
{B,C}	3
{B,D}	4
{C,D}	3

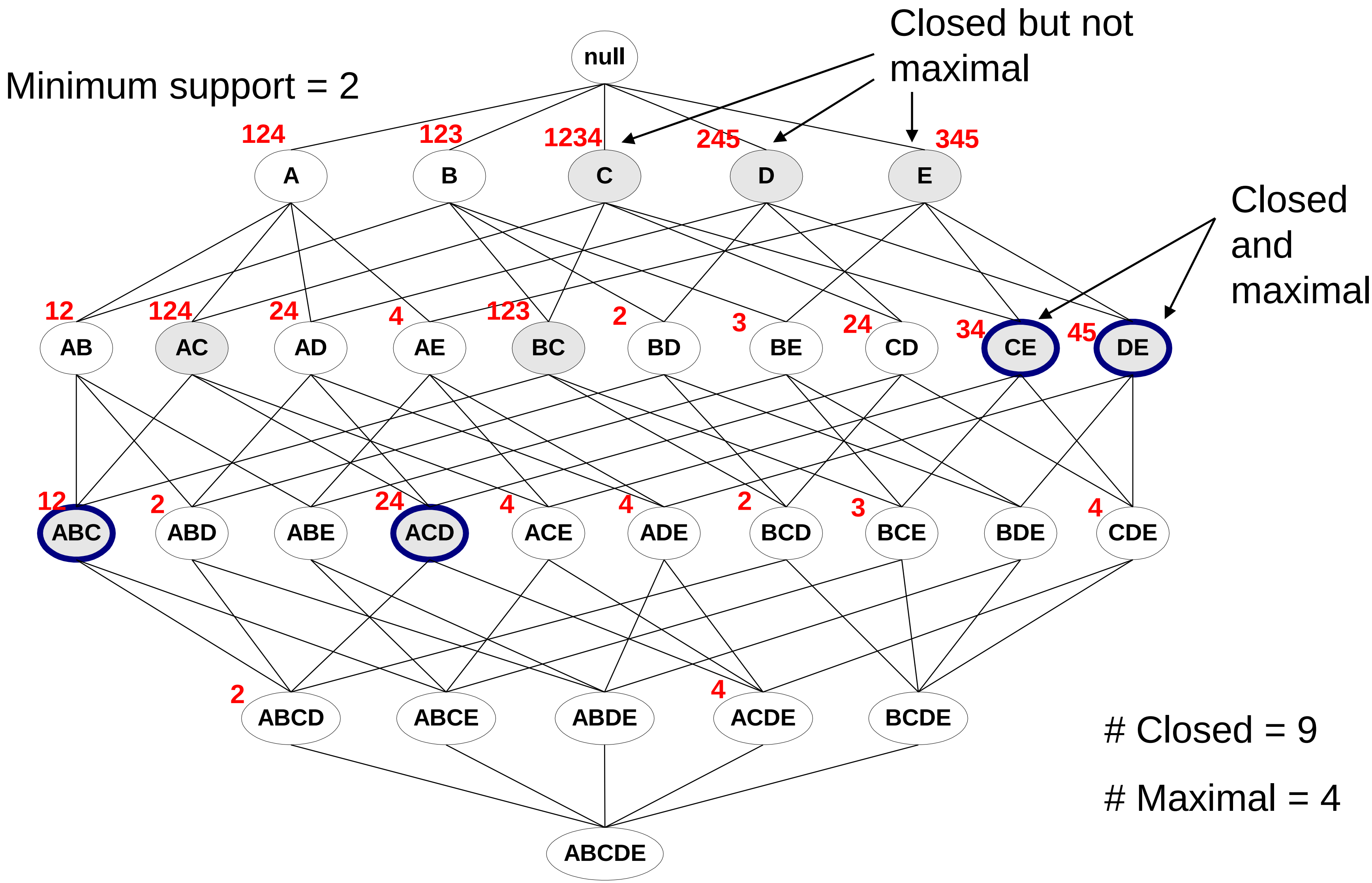
Itemset	Support
{A,B,C}	2
{A,B,D}	3
{A,C,D}	2
{B,C,D}	3
{A,B,C,D}	2

Maximal vs Closed Itemsets

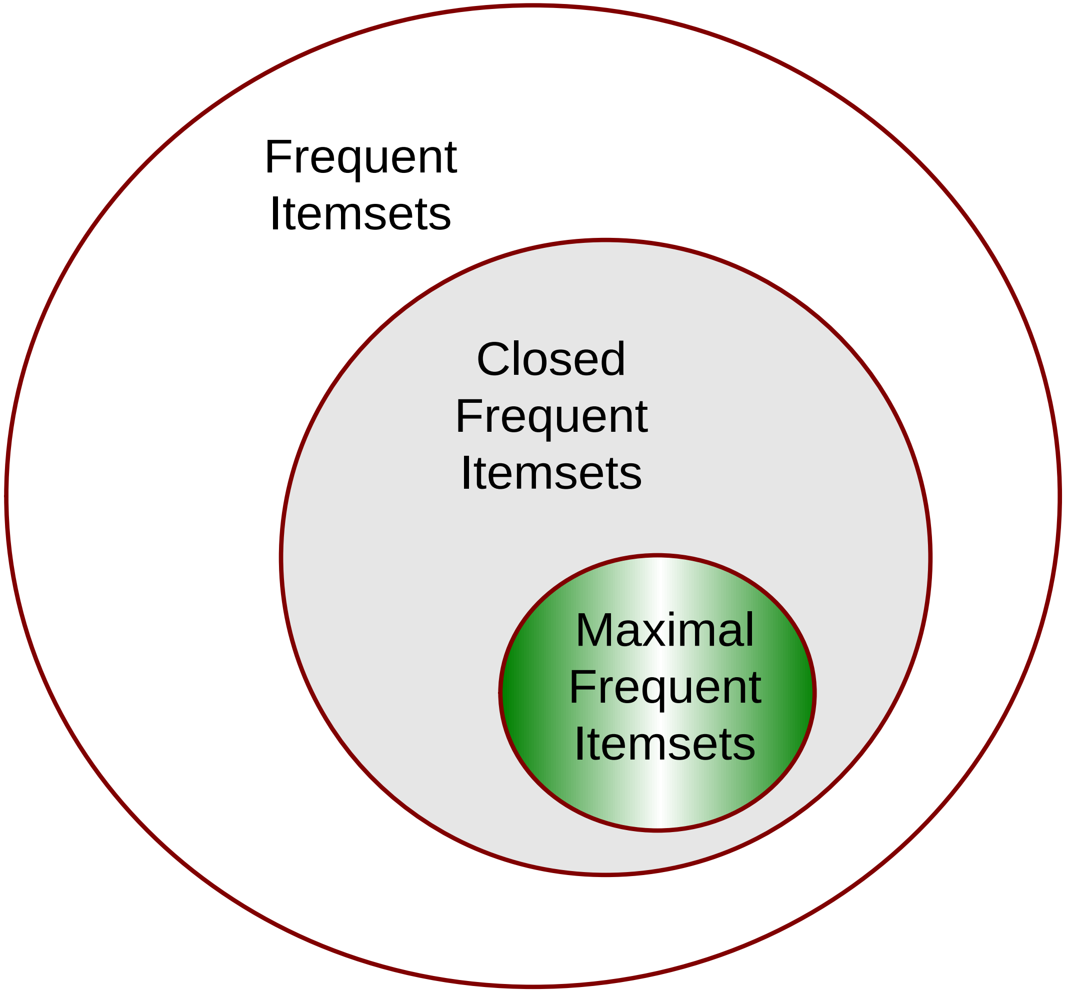
TID	Items
1	ABC
2	ABCD
3	BCE
4	ACDE
5	DE



Maximal vs Closed Frequent Itemsets



Maximal vs Closed Itemsets



Pattern Evaluation

- Association rule algorithms tend to produce too many rules but many of them are uninteresting or redundant
  - Redundant if {A,B,C} $\rightarrow$ {D} and {A,B} $\rightarrow$ {D} have same support & confidence
    - Summarization techniques
  - Uninteresting, if the pattern that is revealed does not offer useful information.
    - Interestingness measures: a hard problem to define

- Interestingness measures can be used to prune/rank the derived patterns
  - Subjective measures: require human analyst
  - Objective measures: rely on the data.

- In the original formulation of association rules, support & confidence are the only measures used

Computing Interestingness Measure

- Given a rule X → Y, information needed to compute rule interestingness can be obtained from a contingency table

Contingency table for X → Y

	Y	$\bar{Y}$	
X	f_{11}	f_{10}	f_{1+}
$\bar{X}$	f_{01}	f_{00}	f_{0+}
	f_{+1}	f_{+0}	N

f_{11}: support of X and Y

f_{10}: support of $\underline{X}$ and $\overline{Y}$

f_{01}: support of $\underline{X}$ and $\underline{Y}$

f_{00}: support of X and Y

X: itemset X appears in tuple

Y: itemset Y appears in tuple

$\bar{X}$: itemset X does not appear in tuple

$\bar{Y}$: itemset Y does not appear in tuple

Used to define various measures

- support, confidence, lift, Gini, J-measure, etc.

Drawback of Confidence

	Coffee	$\overline{\text{Coffee}}$	
Tea	15	5	20
$\overline{\text{Tea}}$	75	5	80
	90	10	100

Number of people that drink tea

Number of people that drink coffee and tea

Number of people that drink coffee but not tea

Number of people that drink coffee

Association Rule: Tea $\rightarrow$ Coffee

Confidence= $P(\text{Coffee}|\text{Tea}) = \frac{15}{20} = 0.75$

but $P(\text{Coffee}) = \frac{90}{100} = 0.9$

- Although confidence is high, rule is misleading
- $P(\text{Coffee}|\overline{\text{Tea}}) = 0.9375$

Statistical Independence

- Population of 1000 students
  - 600 students know how to swim (S)
  - 700 students know how to bike (B)
  - 420 students know how to swim and bike (S,B)

  - $P(S,B) = 420/1000 = 0.42$
  - $P(S) \times P(B) = 0.6 \times 0.7 = 0.42$

  - $P(S,B) = P(S) \times P(B)$ => Statistical independence

Statistical Independence

- Population of 1000 students
  - 600 students know how to swim (S)
  - 700 students know how to bike (B)
  - 500 students know how to swim and bike (S,B)

  - $P(S,B) = 500/1000 = 0.5$
  - $P(S) \times P(B) = 0.6 \times 0.7 = 0.42$

  - $P(S,B) > P(S) \times P(B)$ => Positively correlated

Statistical Independence

- Population of 1000 students
  - 600 students know how to swim (S)
  - 700 students know how to bike (B)
  - 300 students know how to swim and bike (S,B)

  - $P(S,B) = 300/1000 = 0.3$
  - $P(S) \times P(B) = 0.6 \times 0.7 = 0.42$

  - $P(S,B) < P(S) \times P(B)$ => Negatively correlated

Statistical-based Measures

- Measures that take into account statistical dependence
  - Lift/Interest/PMI

$$\text{Lift} = \frac{P(Y|X)}{P(Y)} = \frac{P(X,Y)}{P(X)P(Y)} = \text{Interest}$$

In text mining it is called: Pointwise Mutual Information

  - Piatesky-Shapiro

$$\text{PS} = P(X,Y) - P(X)P(Y)$$

- All these measures measure deviation from independence
  - The higher, the better (why?)

Example: Lift/Interest

	Coffee	$\overline{\text{Coffee}}$	
Tea	15	5	20
$\overline{\text{Tea}}$	75	5	80
	90	10	100

Association Rule: Tea $\rightarrow$ Coffee

Confidence= P(Coffee|Tea) = 0.75

but P(Coffee) = 0.9

$\Rightarrow$ Lift = 0.75/0.9= 0.8333 (< 1, therefore is negatively associated)

= 0.15/(0.9*0.2)

Another Example

	of	the	of, the
Fraction of documents	0.9	0.9	0.8

$$P(of, the) \approx P(of)P(the)$$

If I was creating a document by picking words randomly, (of, the) have more or less the same probability of appearing together by chance

No correlation

	hong	kong	hong, kong
Fraction of documents	0.2	0.2	0.19

$$P(hong, kong) \gg P(hong)P(kong)$$

(hong, kong) have much lower probability to appear together by chance. The two words appear almost always only together

Positive correlation

	obama	karagounis	obama, karagounis
Fraction of documents	0.2	0.2	0.001

$$P(obama, karagounis) \ll P(obama)P(karagounis)$$

(obama, karagounis) have much higher probability to appear together by chance. The two words appear almost never together

Negative correlation

Drawbacks of Lift/Interest/Mutual Information

	honk	konk	honk, konk
Fraction of documents	0.0001	0.0001	0.0001

$$MI(honk, konk) = \frac{0.0001}{0.0001 * 0.0001} = 10000$$

	hong	kong	hong, kong
Fraction of documents	0.2	0.2	0.19

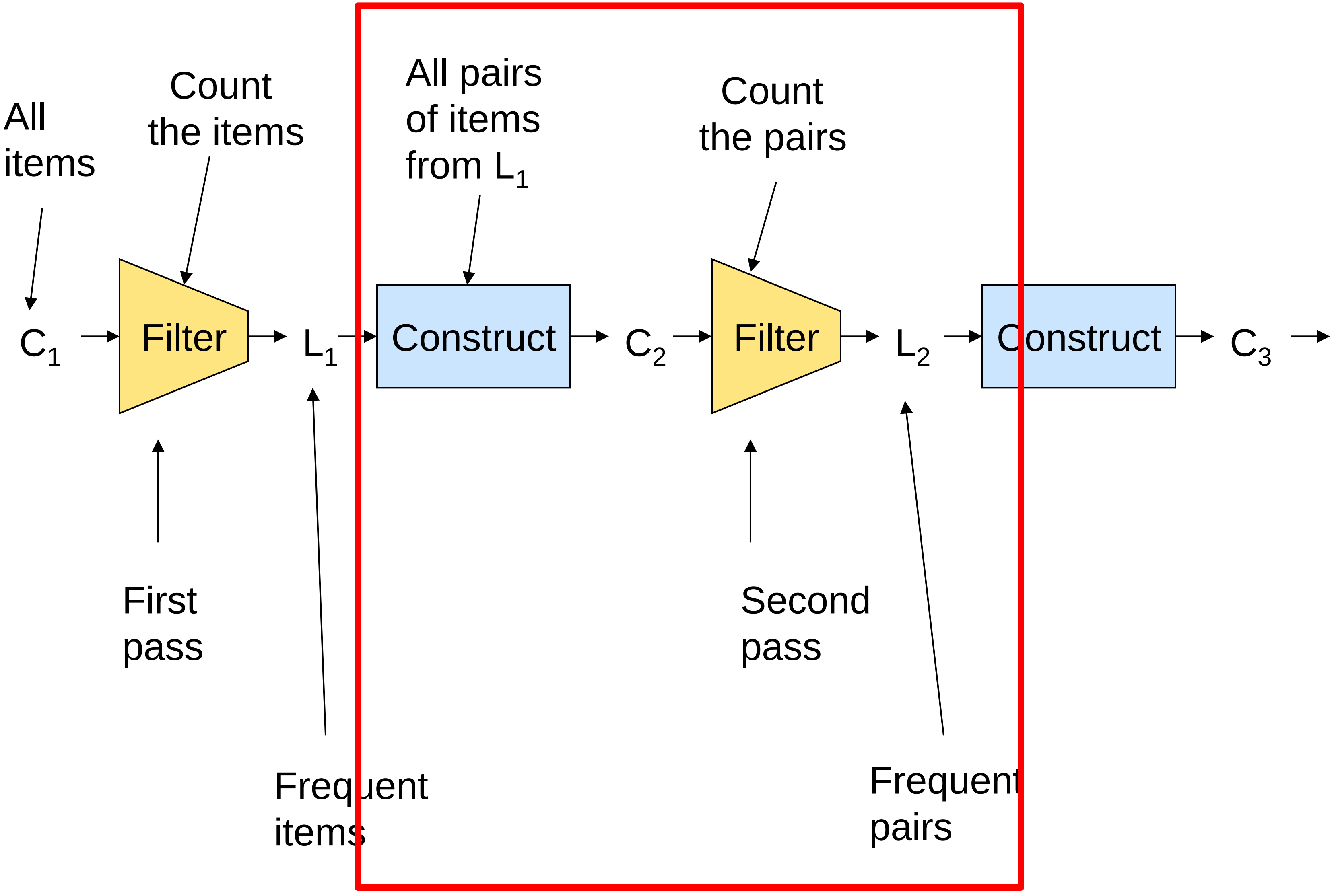
$$MI(hong, kong) = \frac{0.19}{0.2 * 0.2} = 4.75$$

Rare co-occurrences are deemed more interesting.
But this is not always what we want

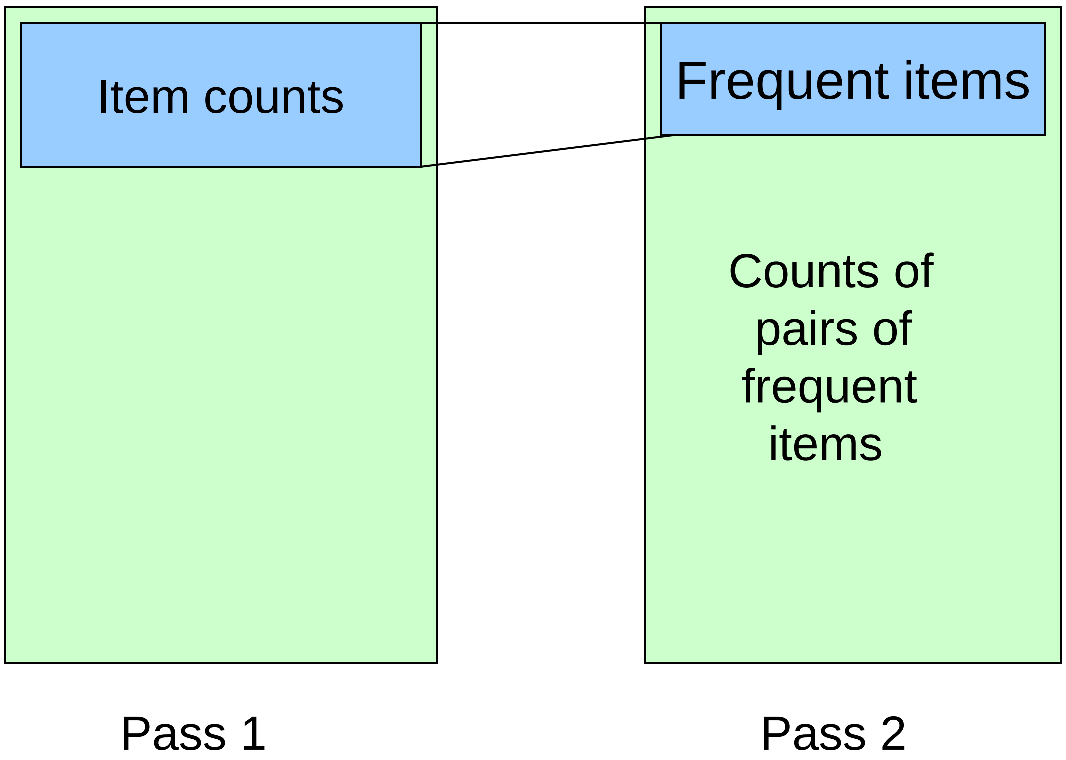
ALTERNATIVE FREQUENT ITEMSET COMPUTATION

Slides taken from Mining Massive Datasets course by Anand Rajaraman and Jeff Ullman.

Finding the frequent pairs is usually the most expensive operation

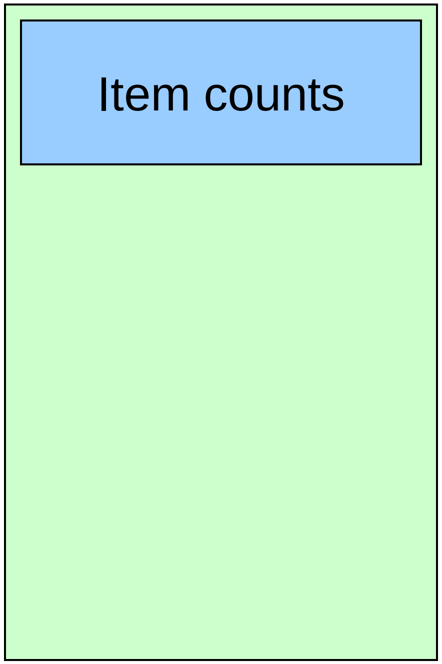


Picture of A-Priori



PCY Algorithm

- During Pass 1 (computing frequent items) of Apriori, most memory is idle.
- Use that memory to keep a hash table where pairs of items are hashed.
- The hash table keeps just counts of the number of pairs hashed in each bucket, not the pairs themselves.



Pass 1

Needed Extensions

1. Pairs of items need to be generated from the input file; they are not present in the file.
2. We are not just interested in the presence of a pair, but we need to see whether it is present at least *s* (support) times.

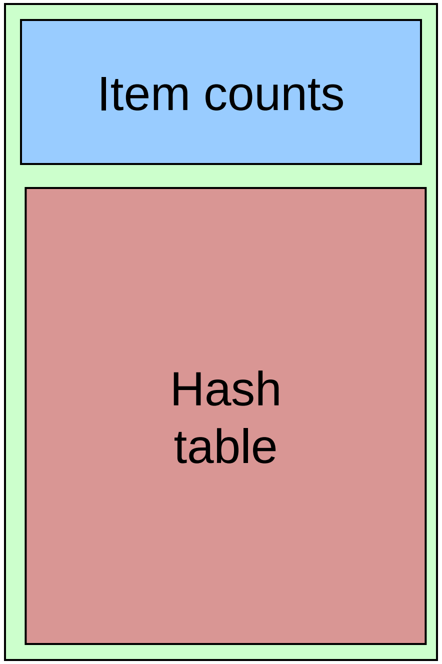
PCY Algorithm – (2)

- A bucket is frequent if its count is at least the support threshold.
- If a bucket is not frequent, no pair that hashes to that bucket could possibly be a frequent pair.
  - The opposite is not true, a bucket may be frequent but hold infrequent pairs
- On Pass 2 (frequent pairs), we only count pairs that hash to frequent buckets.

PCY Algorithm – Before Pass 1 Organize Main Memory

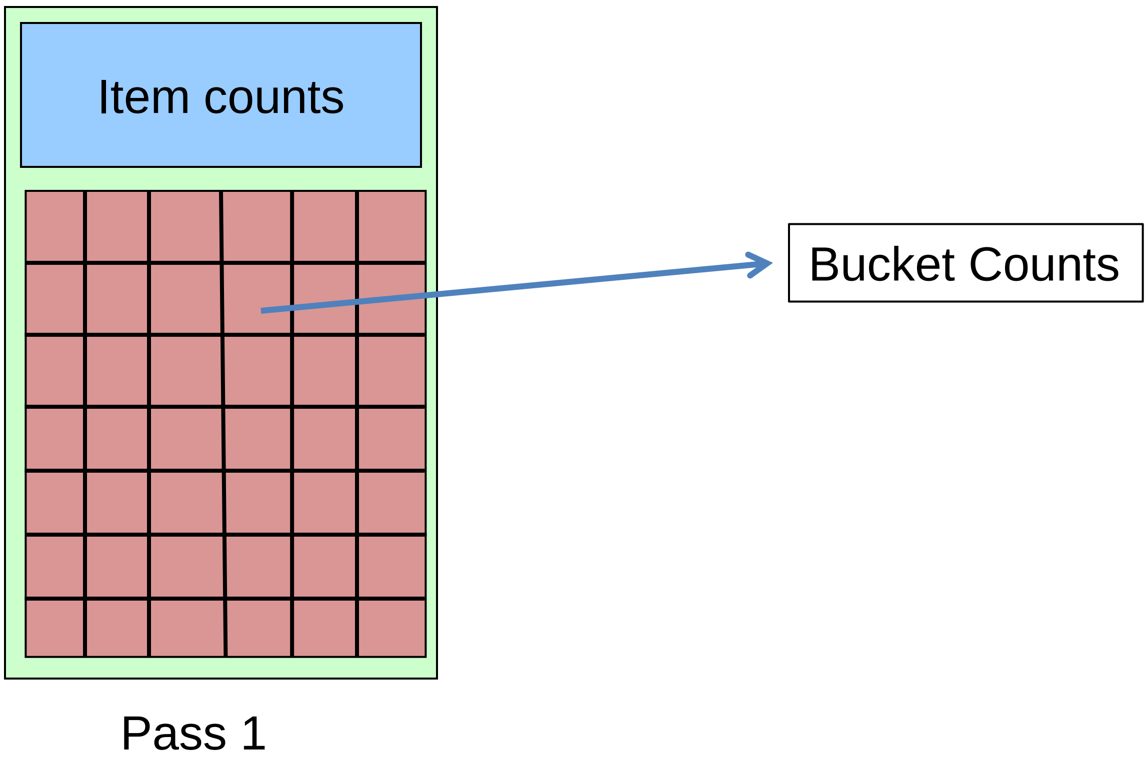
- Space to count each item.
  - One (typically) 4-byte integer per item.
- Use the rest of the space for as many integers, representing buckets, as we can.

Picture of PCY



Pass 1

Picture of PCY



PCY Algorithm – Pass 1

```
FOR (each basket) {
    FOR (each item in the basket)
      add 1 to item's count;
    FOR (each pair of items in the basket)
    {
      hash the pair to a bucket;
      add 1 to the count for that bucket
    }
}
```

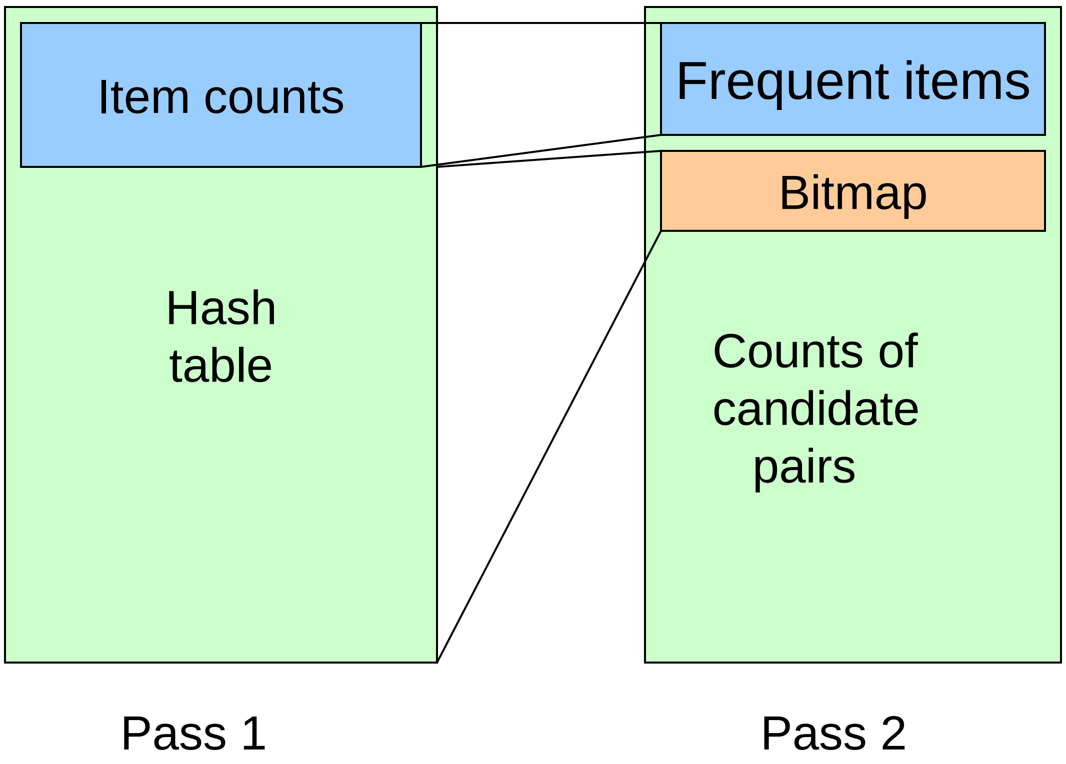
Observations About Buckets

1. A bucket that a frequent pair hashes to is surely frequent.
   - We cannot use the hash table to eliminate any member of this bucket.
2. Even without any frequent pair, a bucket can be frequent.
   - Again, nothing in the bucket can be eliminated.
3. But in the best case, the count for a bucket is less than the support *s*.
   - Now, all pairs that hash to this bucket can be eliminated as candidates, even if the pair consists of two frequent items.

PCY Algorithm – Between Passes

- Replace the buckets by a bit-vector:
  - 1 means the bucket is frequent; 0 means it is not.
- 4-byte integers are replaced by bits, so the bit-vector requires 1/32 of memory.
- Also, find which items are frequent and list them for the second pass.
  - Same as with Apriori

Picture of PCY



PCY Algorithm – Pass 2

- Count all pairs {*i*, *j* } that meet the conditions for being a candidate pair:
  1. Both *i* and *j* are frequent items.
  2. The pair {*i*, *j* }, hashes to a bucket number whose bit in the bit vector is 1.

- Notice both these conditions are necessary for the pair to have a chance of being frequent.

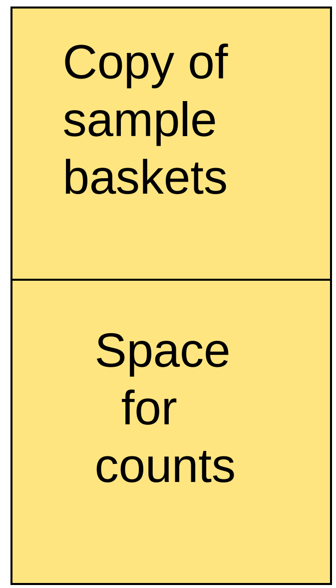
All (Or Most) Frequent Itemsets in less than 2 Passes

- A-Priori, PCY, etc., take *k* passes to find frequent itemsets of size *k*.
- Other techniques use 2 or fewer passes for all sizes:
  - Simple sampling algorithm.
  - SON (Savasere, Omiecinski, and Navathe).
  - Toivonen.

Simple Sampling Algorithm – (1)

- Take a random sample of the market baskets.
- Run Apriori or one of its improvements (for sets of all sizes, not just pairs) in main memory, so you don't pay for disk I/O each time you increase the size of itemsets.
  - Make sure the sample is such that there is enough space for counts.

Main-Memory Picture



Simple Algorithm – (2)

- Use as your support threshold a suitable, scaled-back number.
  - E.g., if your sample is 1/100 of the baskets, use *s* /100 as your support threshold instead of *s*.
- You could stop here (single pass)
  - What could be the problem?

Simple Algorithm – Option

- Optionally, verify that your guesses are truly frequent in the entire data set by a second pass (eliminate false positives)

- But you don't catch sets frequent in the whole but not in the sample. (false negatives)
  - Smaller threshold, e.g., s /125, helps catch more truly frequent itemsets.
    - But requires more space.

SON Algorithm – (1)

- First pass: Break the data into chunks that can be processed in main memory.
- Read one chunk at the time
  - Find all frequent itemsets for each chunk.
  - Threshold = s/number of chunks

- An itemset becomes a candidate if it is found to be frequent in any one or more chunks of the baskets.

SON Algorithm – (2)

- Second pass: count all the candidate itemsets and determine which are frequent in the entire set.

- Key “monotonicity” idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.
    - Why?

SON Algorithm – Distributed Version

- This idea lends itself to distributed data mining.
- If baskets are distributed among many nodes, compute frequent itemsets at each node, then distribute the candidates from each node.
- Finally, accumulate the counts of all candidates.

Toivonen's Algorithm – (1)

- Start as in the simple sampling algorithm, but lower the threshold slightly for the sample.
  - Example: if the sample is 1% of the baskets, use $s/125$ as the support threshold rather than $s/100$.
  - Goal is to avoid missing any itemset that is frequent in the full set of baskets.

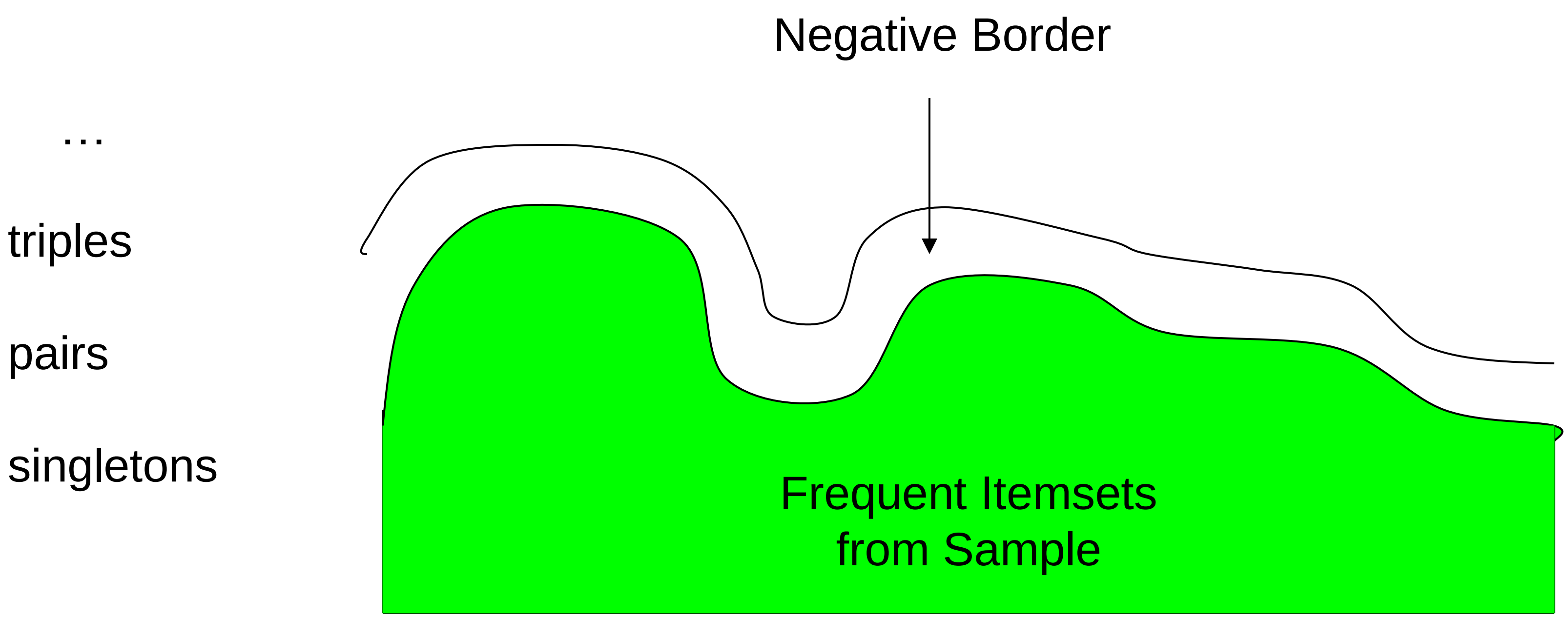
Toivonen's Algorithm – (2)

- Add to the itemsets that are frequent in the sample the negative border of these itemsets.
- An itemset is in the negative border if it is not deemed frequent in the sample, but all its immediate subsets are.

Reminder: Negative Border

- *ABCD* is in the negative border if and only if:
  1. It is not frequent in the sample, but
  2. All of *ABC, BCD, ACD*, and *ABD* are.
- *A* is in the negative border if and only if it is not frequent in the sample.
  - Because the empty set is always frequent.
    - Unless there are fewer baskets than the support threshold (silly case).

Picture of Negative Border



Toivonen's Algorithm – (3)

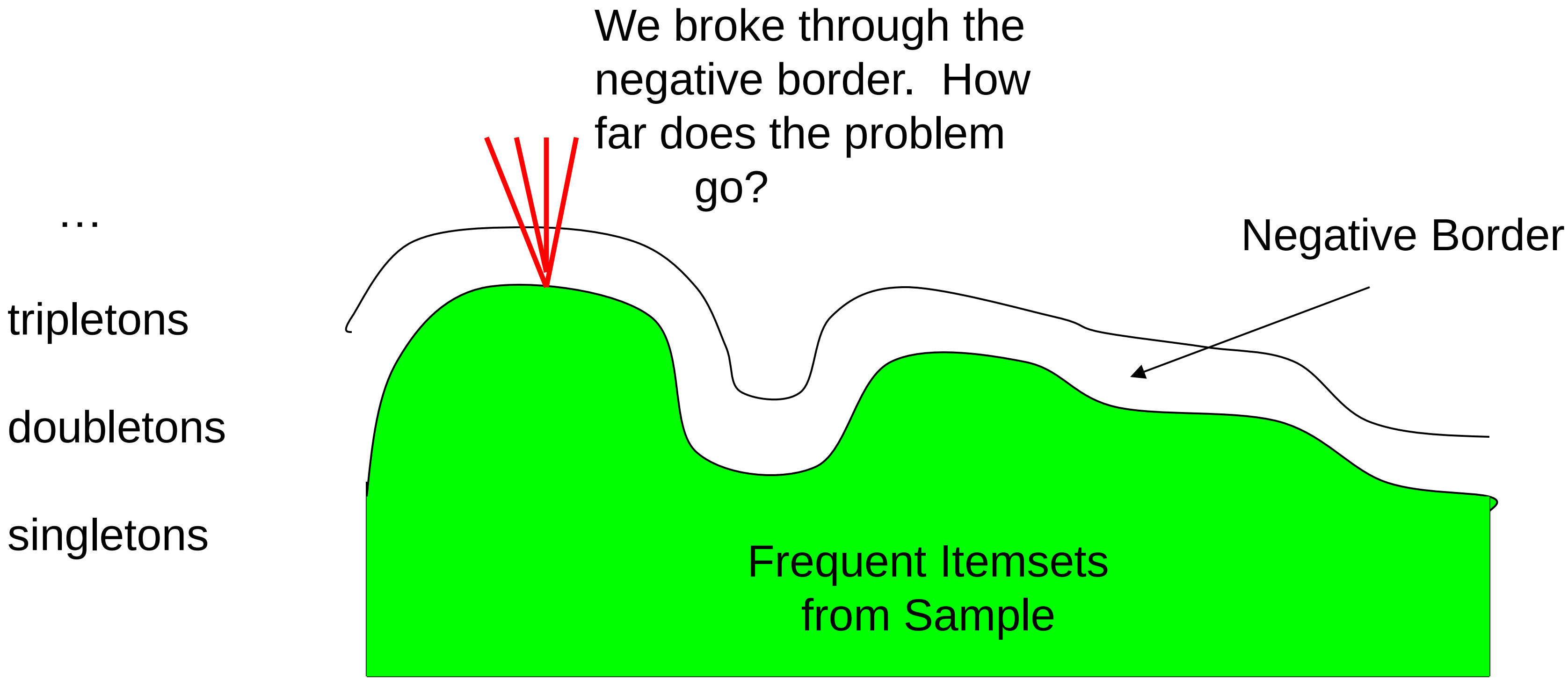
- In a second pass, count all candidate frequent itemsets from the first pass, and also count their negative border.
- If no itemset from the negative border turns out to be frequent, then the candidates found to be frequent in the whole data are exactly the frequent itemsets.

Toivonen's Algorithm – (4)

- What if we find that something in the negative border is actually frequent?
  - We must start over again!

- Try to choose the support threshold so the probability of failure is low, while the number of itemsets checked on the second pass fits in main-memory.

If Something in the Negative Border is Frequent . . .



Theorem:

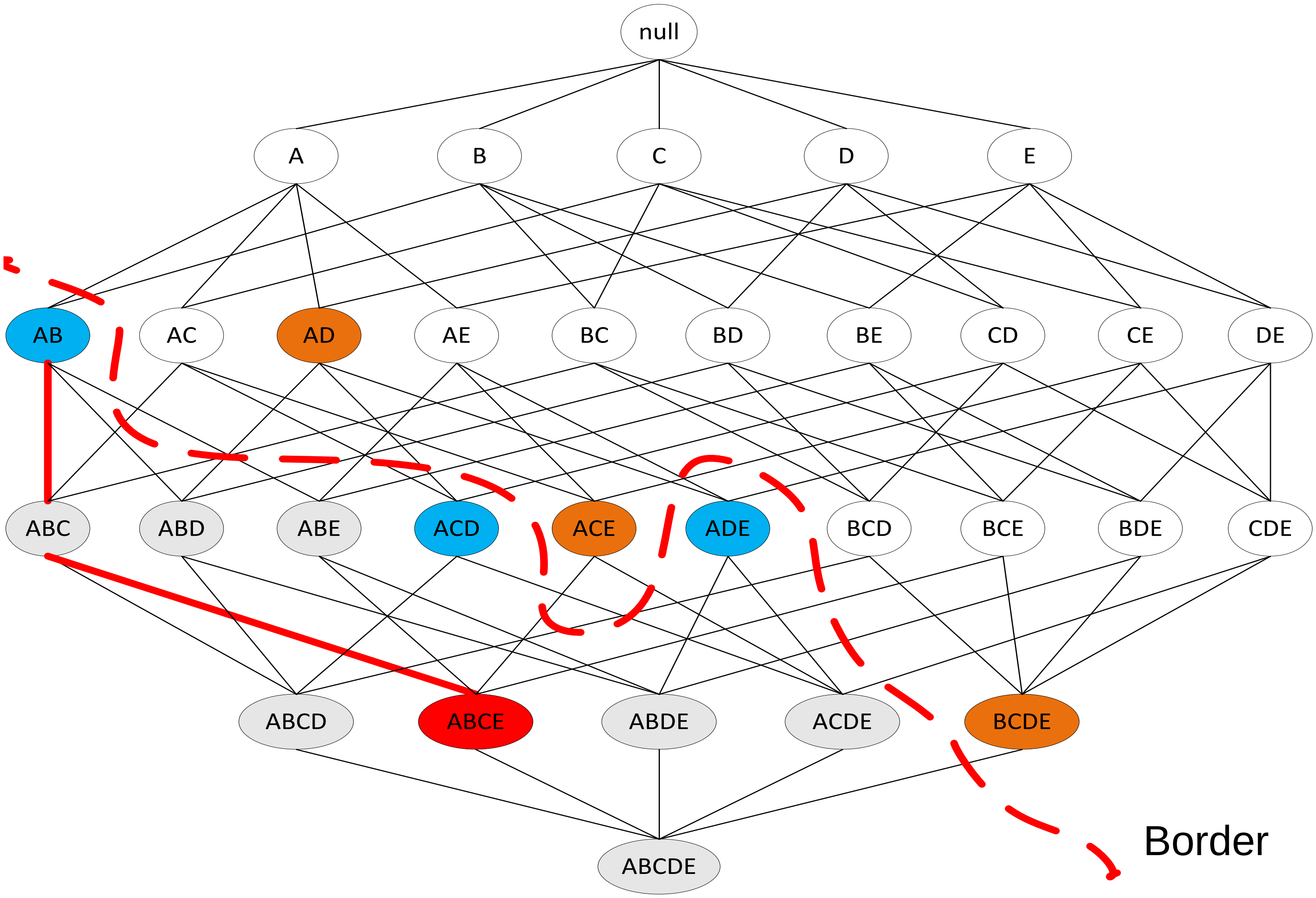
- If there is an itemset that is frequent in the whole, but not frequent in the sample, then there is a member of the negative border for the sample that is frequent in the whole.

Proof: Suppose not; i.e.;

1. There is an itemset *S* frequent in the whole but not frequent in the sample, and
2. Nothing in the negative border is frequent in the whole.

- Let *T* be a smallest subset of *S* that is not frequent in the sample.
- *T* is frequent in the whole (*S* is frequent + monotonicity).
- *T* is in the negative border (else not “smallest”).

Example


null
A
B
C
D
E
AB
AC
AD
AE
BC
BD
BE
CD
CE
DE
ABC
ABD
ABE
ACD
ACE
ADE
BCD
BCE
BDE
CDE
ABCD
ABCE
ABDE
ACDE
BCDE
ABCDE
Border

THE FP-TREE AND THE FP-GROWTH ALGORITHM

Slides from course lecture of E. Pitoura

Overview

- The FP-tree contains a compressed representation of the transaction database.
  - A trie (prefix-tree) data structure is used
  - Each transaction is a path in the tree – paths can overlap.

- Once the FP-tree is constructed the recursive, divide-and-conquer FP-Growth algorithm is used to enumerate all frequent itemsets.

FP-tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

- The FP-tree is a trie (prefix tree)
- Since transactions are sets of items, we need to transform them into ordered sequences so that we can have prefixes
  - Otherwise, there is no common prefix between sets {A,B} and {B,C,A}
- We need to impose an order to the items
  - Initially, assume a lexicographic order.

FP-tree Construction

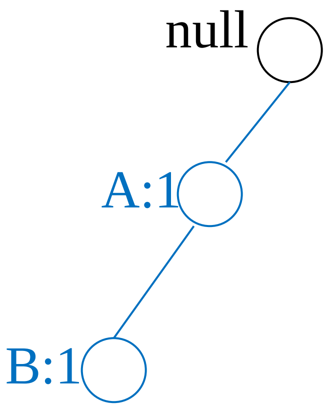
- Initially the tree is empty

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

FP-tree Construction

- Reading transaction TID = 1

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



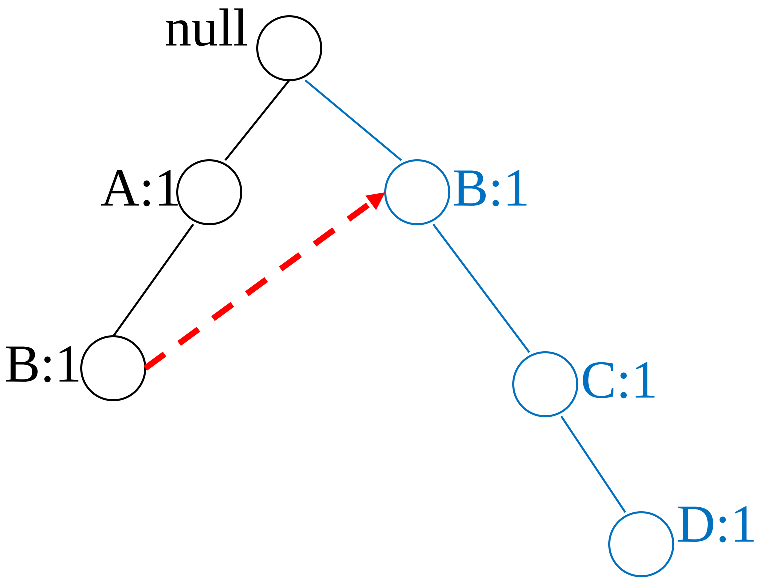
Node label = item:support

- Each node in the tree has a label consisting of the item and the support (number of transactions that reach that node, i.e. follow that path)

FP-tree Construction

- Reading transaction TID = 2

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}



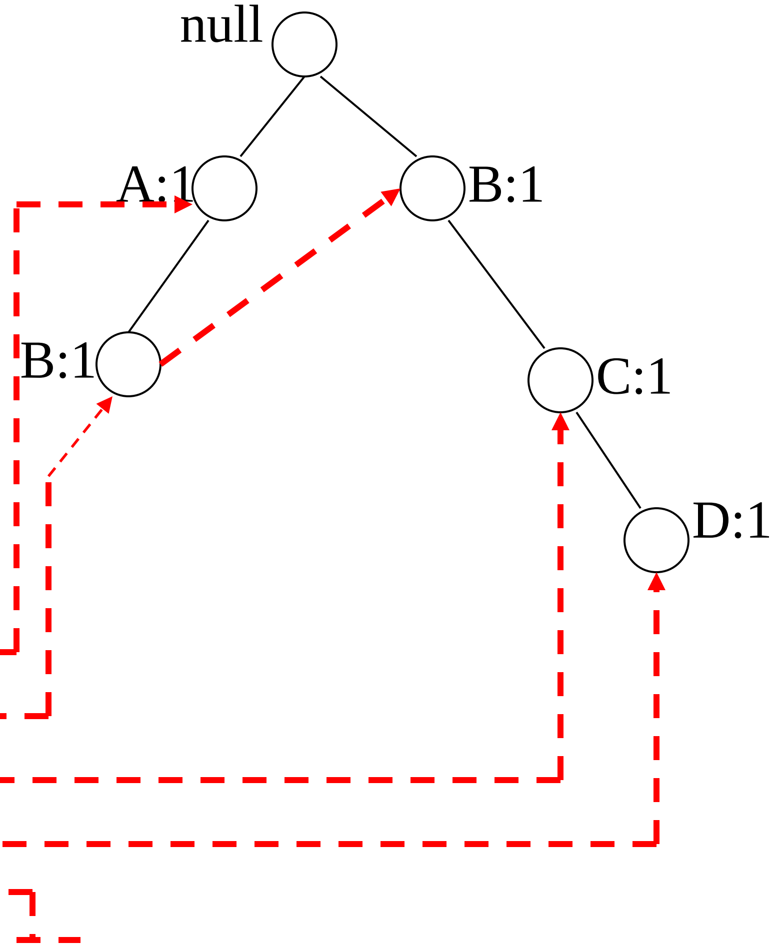
Each transaction is a path in the tree

- We add pointers between nodes that refer to the same item

FP-tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

After reading transactions TID=1, 2:



Header Table

Item	Pointer
A	
B	
C	
D	
E	

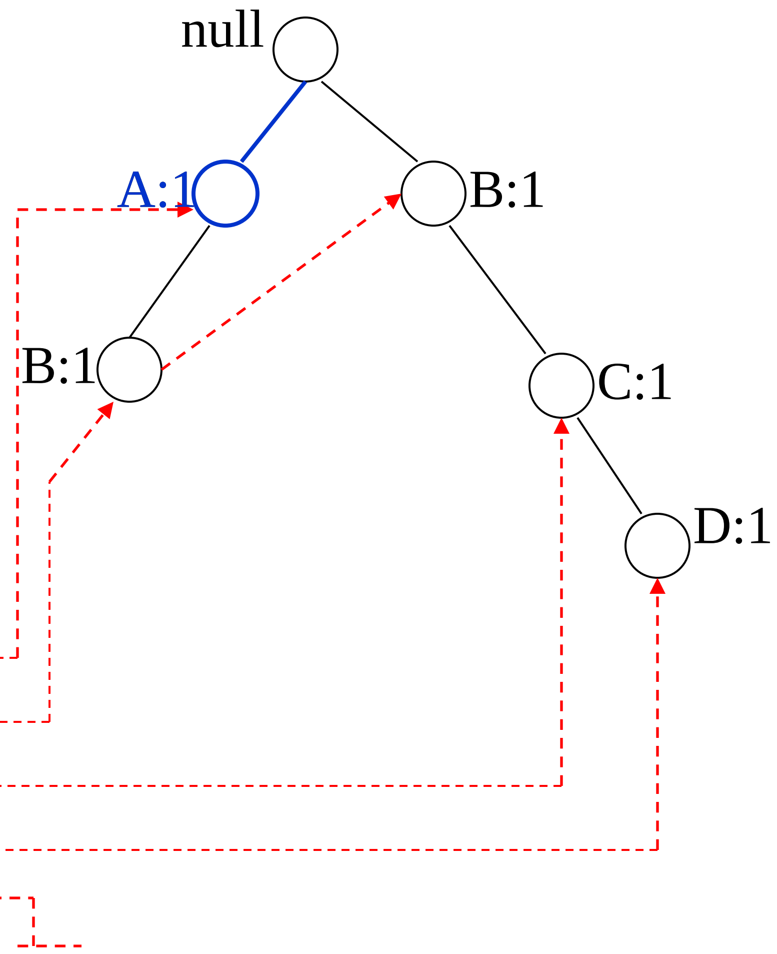
The Header Table and the pointers assist in computing the itemset support

FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Item	Pointer
A	
B	
C	
D	
E	

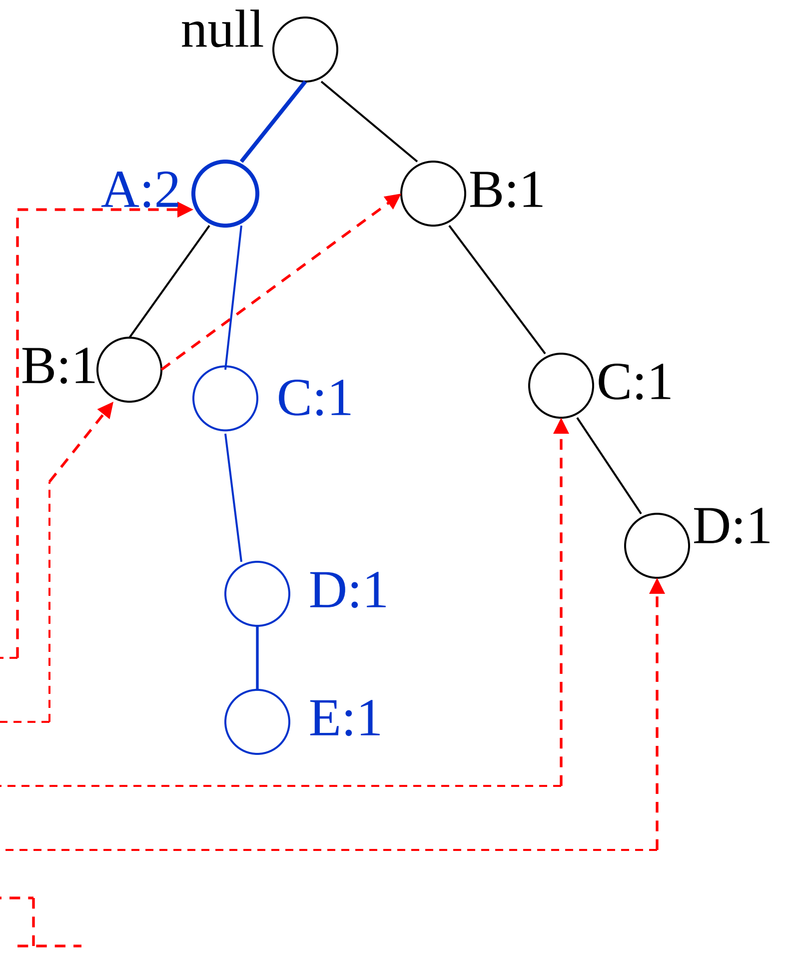


FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Item	Pointer
A	
B	
C	
D	
E	

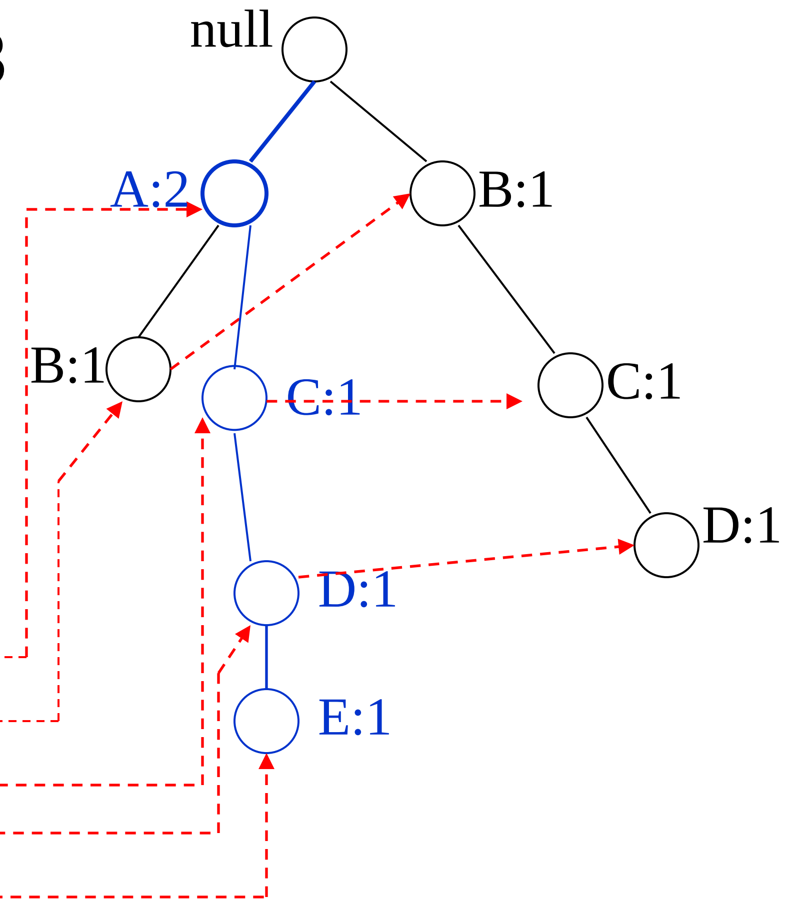


FP-tree Construction

- Reading transaction TID = 3

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Item	Pointer
A	
B	
C	
D	
E	



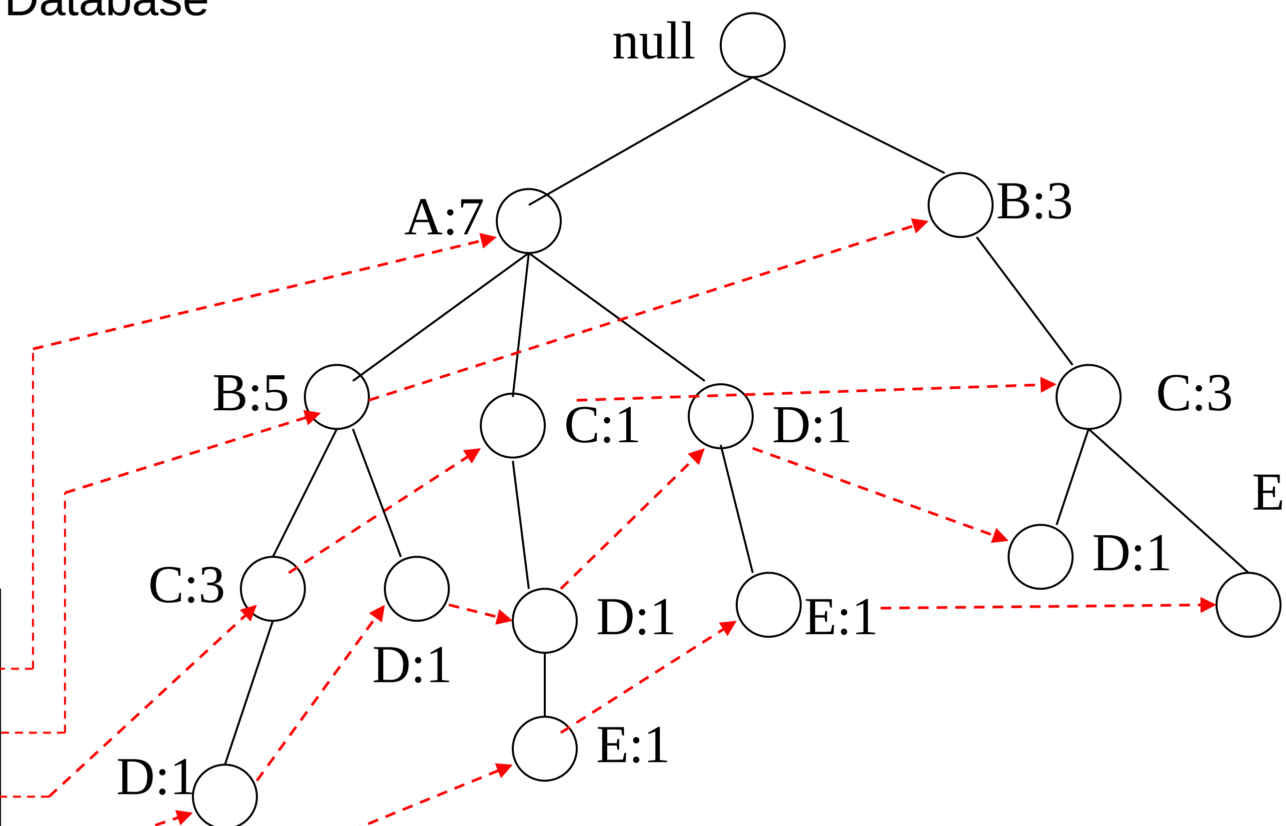
Each transaction is a path in the tree

FP-Tree Construction

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Transaction Database

Each transaction is a path in the tree



Header table

Item	Pointer
A	
B	
C	
D	
E	

Pointers are used to assist frequent itemset generation

FP-tree size

- Every transaction is a path in the FP-tree
- The size of the tree depends on the compressibility of the data
  - Extreme case: All transactions are the same, the FP-tree is a single branch
  - Extreme case: All transactions are different the size of the tree is the same as that of the database (bigger actually since we need additional pointers)

Item ordering

- The size of the tree also depends on the ordering of the items.
- Heuristic: order the items according to their frequency from larger to smaller.
  - We would need to do an extra pass over the dataset to count frequencies
- Example:

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

σ(A)=7, σ(B)=8, σ(C)=7, σ(D)=5, σ(E)=3

Ordering : B,A,C,D,E

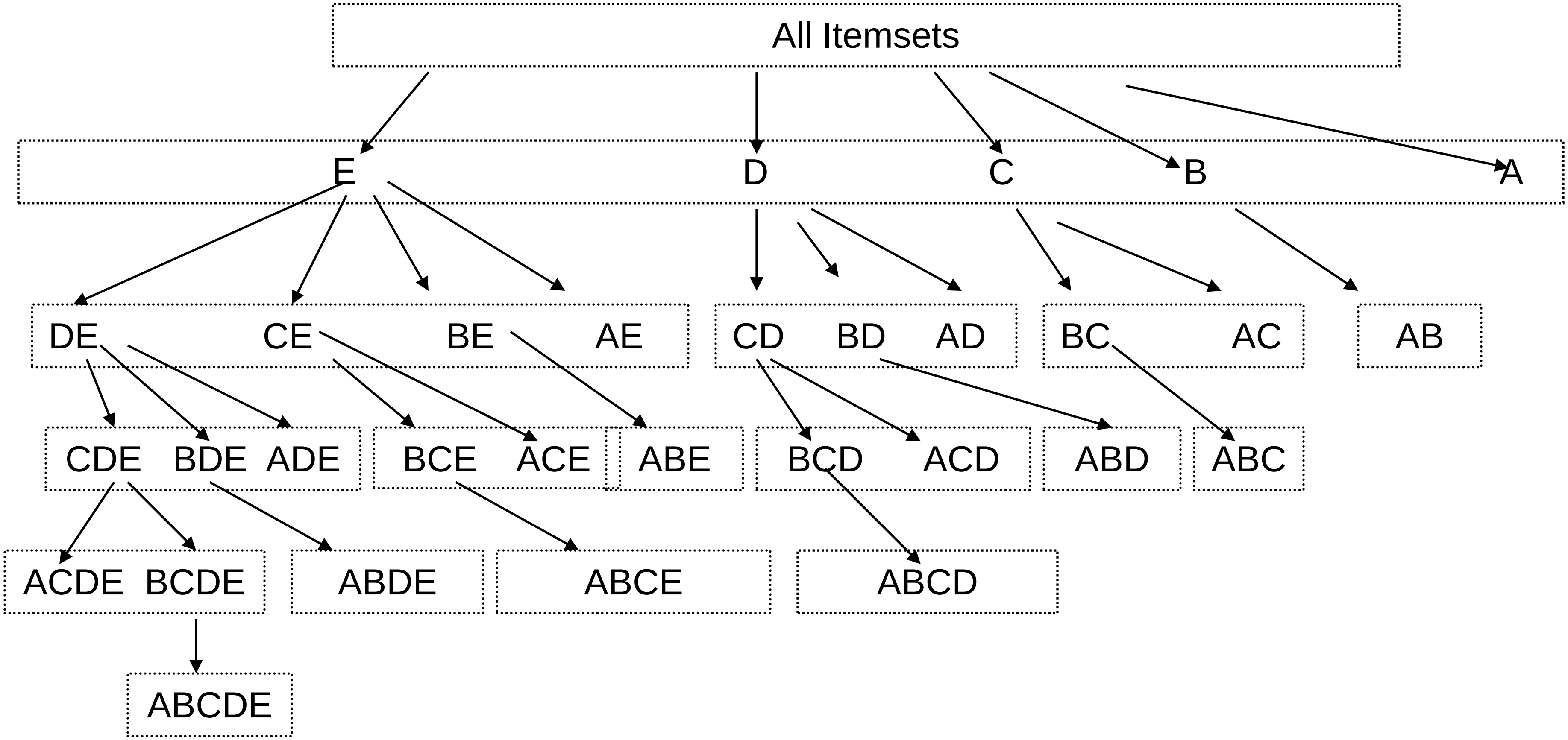
→

TID	Items
1	{B,A}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{B,A,C}
6	{B,A,C,D}
7	{B,C}
8	{B,A,C}
9	{B,A,D}
10	{B,C,E}

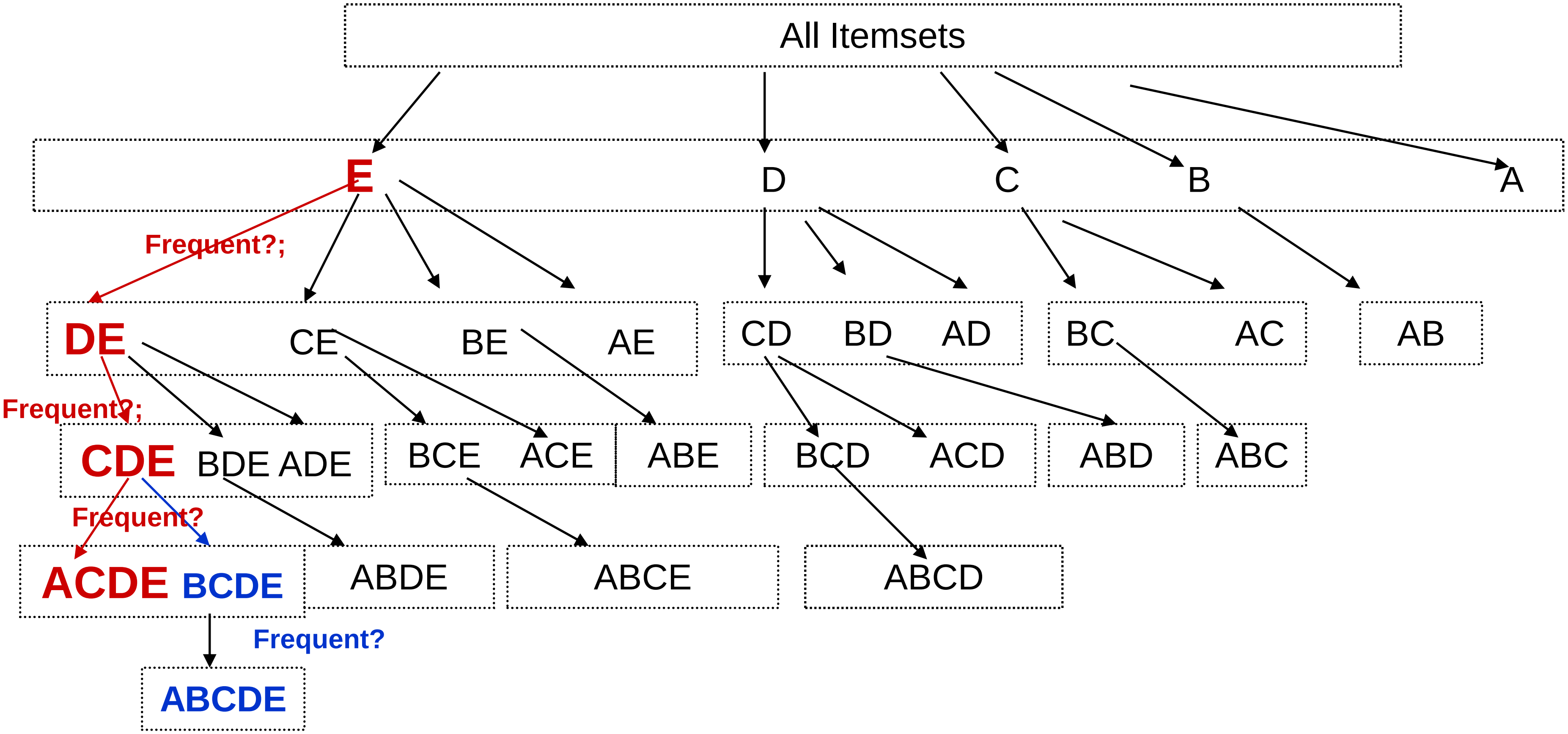
Finding Frequent Itemsets

- Input: The FP-tree
- Output: All Frequent Itemsets and their support
- Method:
  - Divide and Conquer:
  - Consider all itemsets that end in: E, D, C, B, A
    - For each possible ending item, consider the itemsets with last items one of items preceding it in the ordering
    - E.g, for E, consider all itemsets with last item D, C, B, A. This way we get all the itesets ending at DE, CE, BE, AE
    - Proceed recursively this way.
    - Do this for all items.

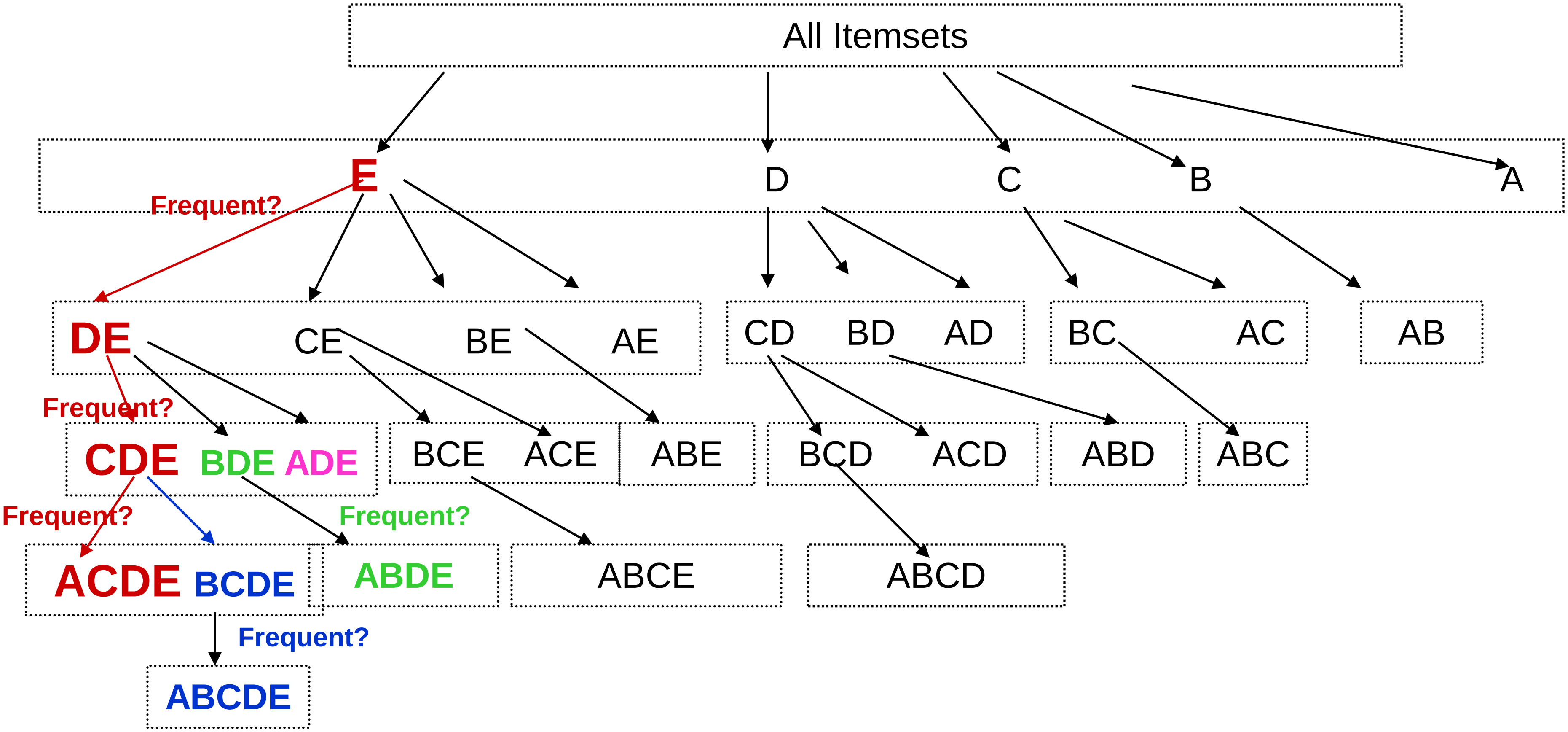
Frequent itemsets



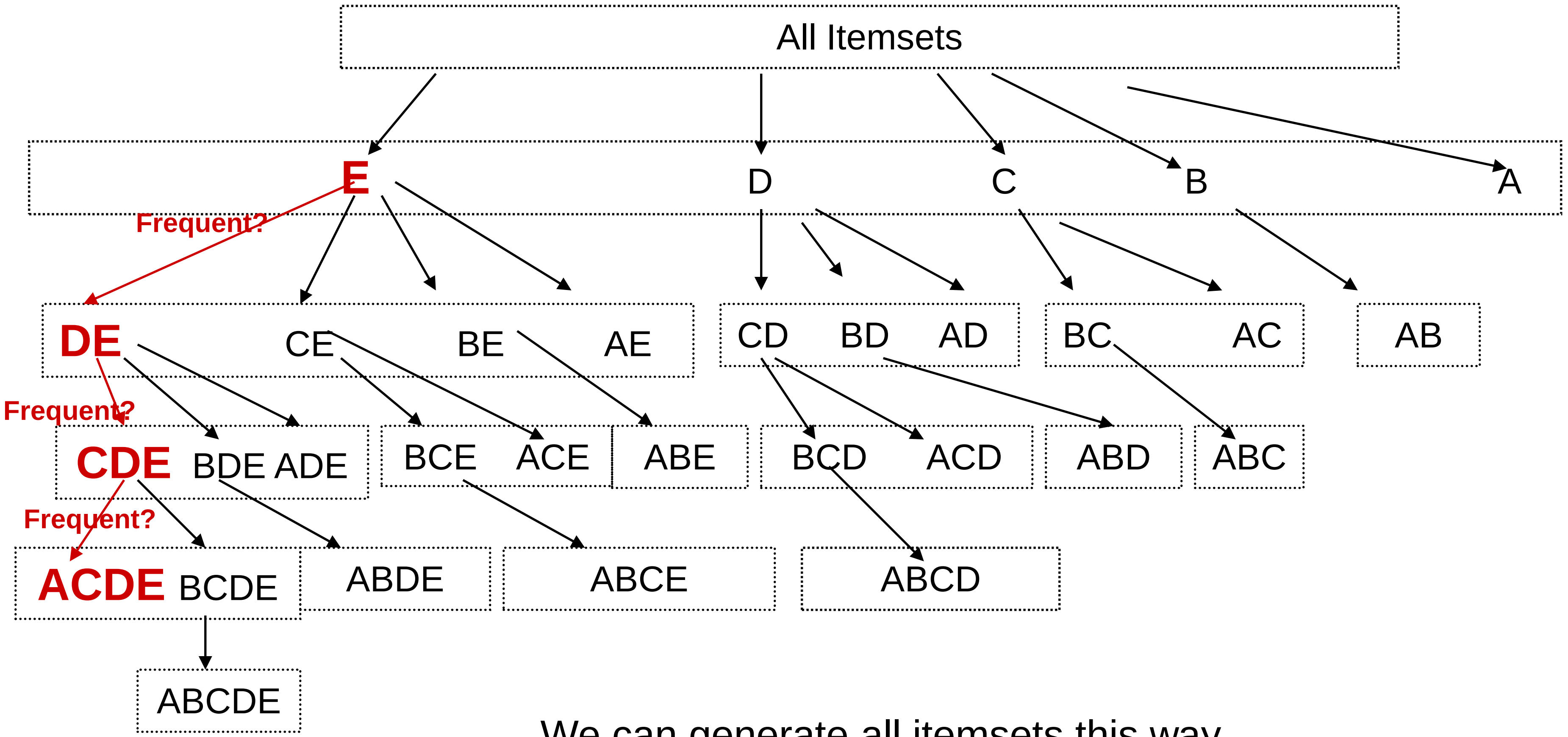
Frequent Itemsets



Frequent Itemsets



Frequent Itemsets



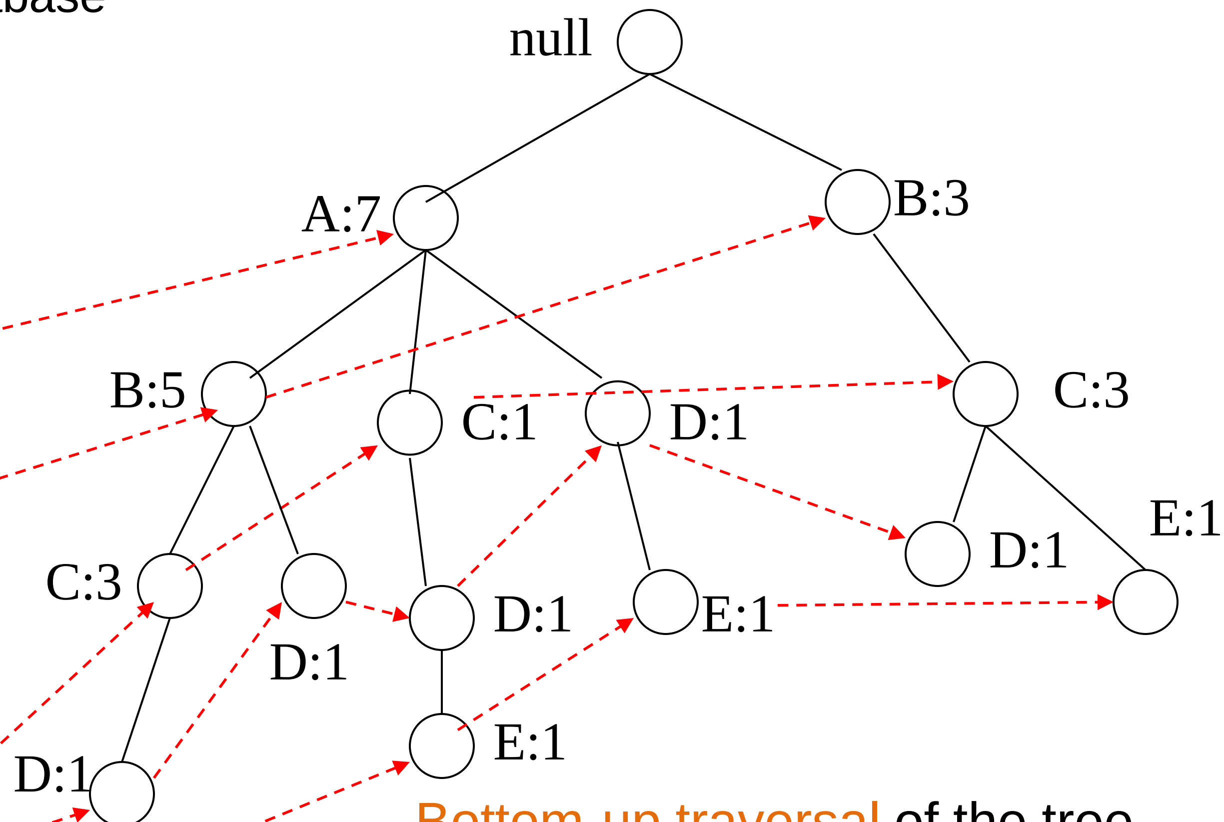
We can generate all itemsets this way

We expect the FP-tree to contain a lot less

Using the FP-tree to find frequent itemsets

TID	Items
1	{A,B}
2	{B,C,D}
3	{A,C,D,E}
4	{A,D,E}
5	{A,B,C}
6	{A,B,C,D}
7	{B,C}
8	{A,B,C}
9	{A,B,D}
10	{B,C,E}

Transaction Database



Header table

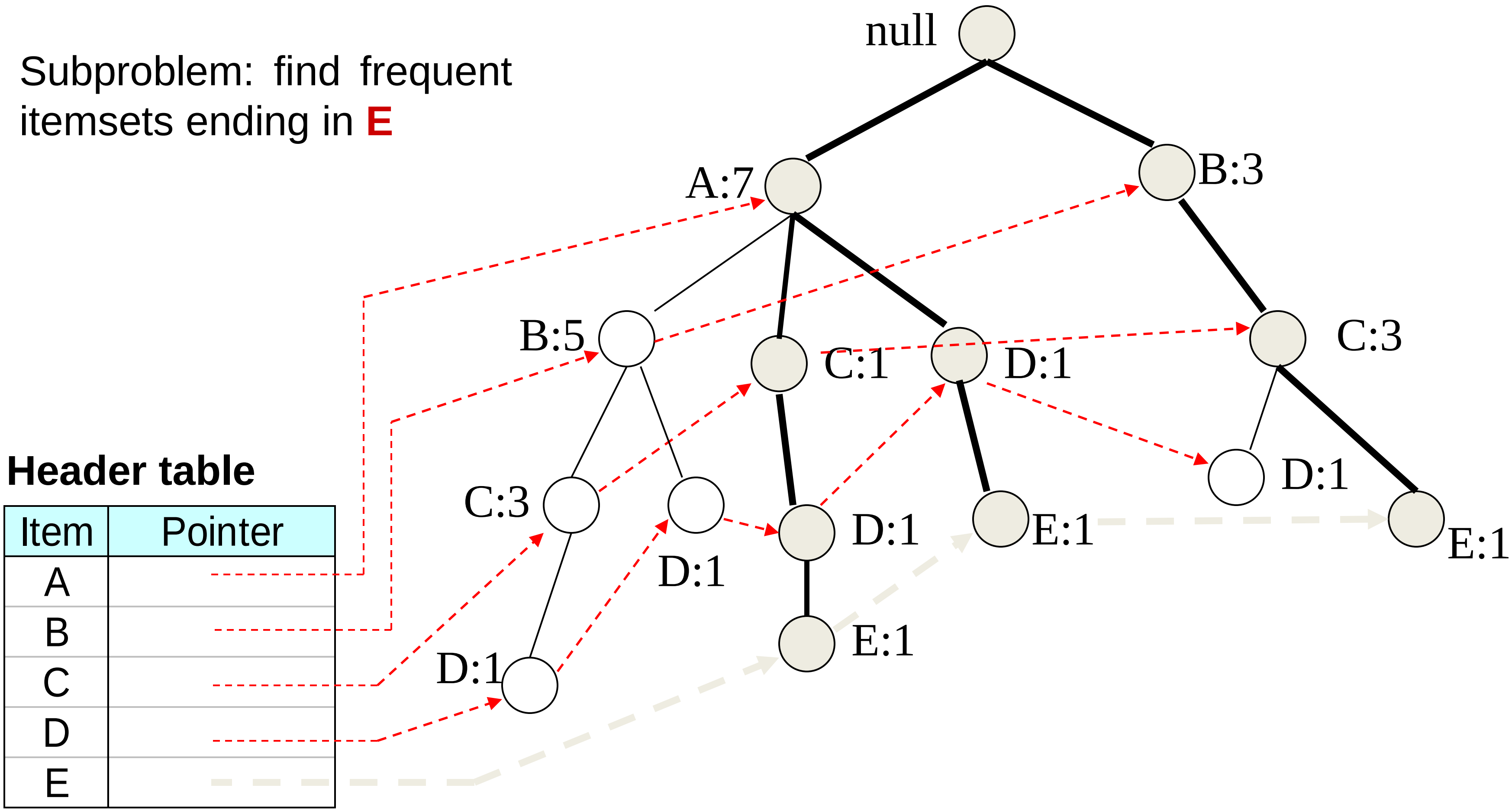
Item	Pointer
A	
B	
C	
D	
E	

Bottom-up traversal of the tree.

First, itemsets ending in E, then D, etc, each time a suffix-based class

Finding Frequent Itemsets

Subproblem: find frequent itemsets ending in **E**



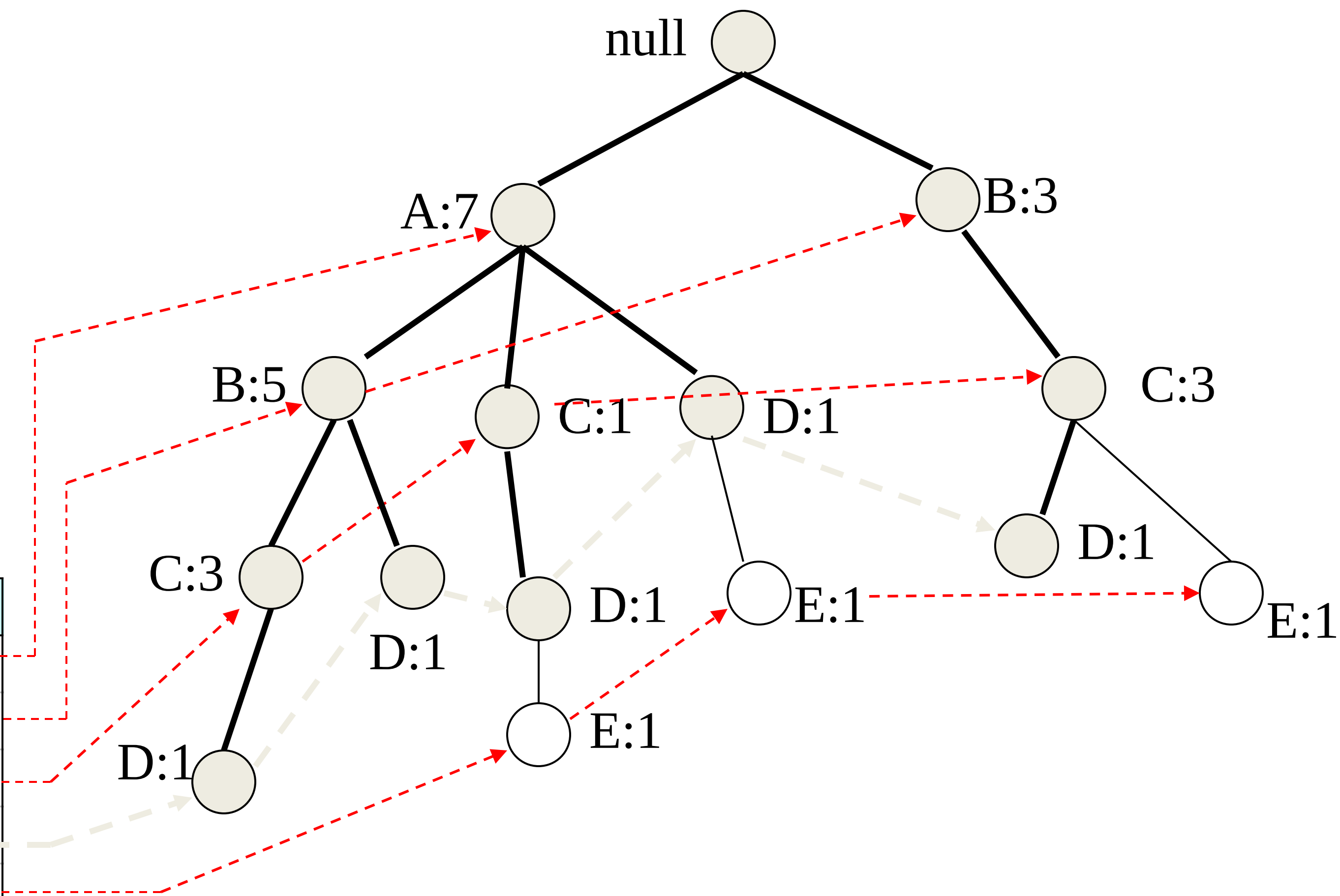
- We will then see how to compute the support for the possible itemsets

Finding Frequent Itemsets

Ending in **D**

Header table

Item	Pointer
A	
B	
C	
D	
E	

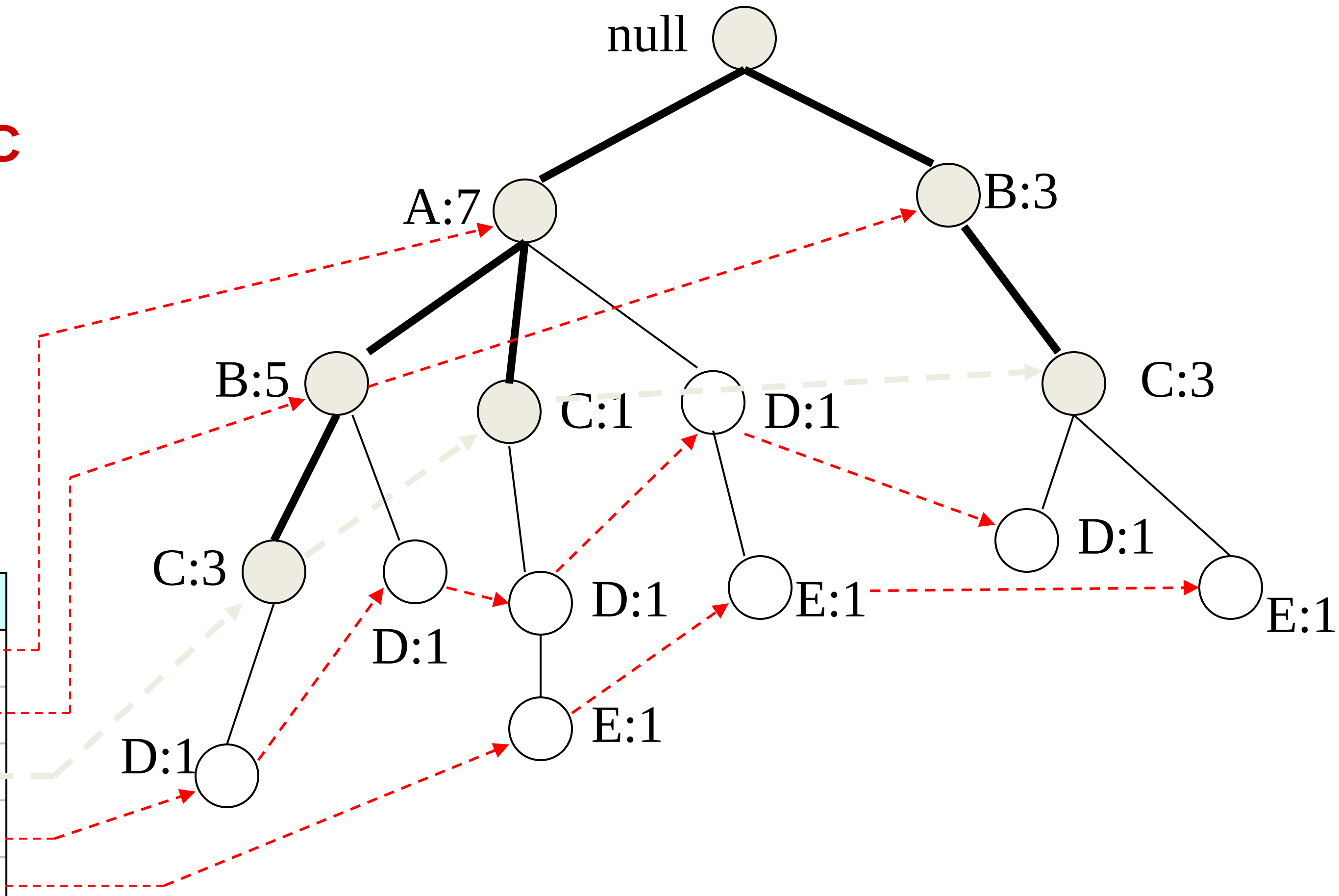


Finding Frequent Itemsets

Ending in **C**

Header table

Item	Pointer
A	
B	
C	
D	
E	

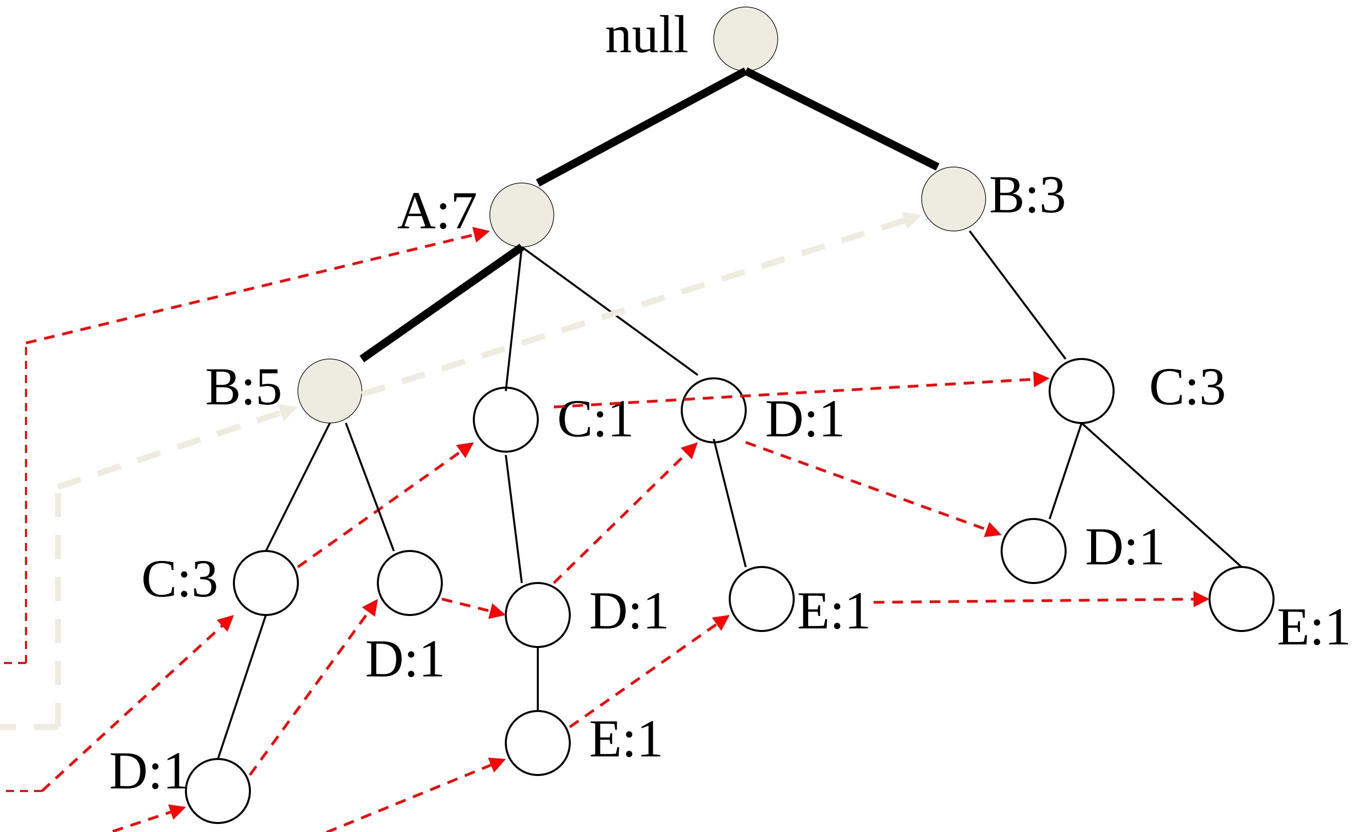


Finding Frequent Itemsets

Ending in **B**

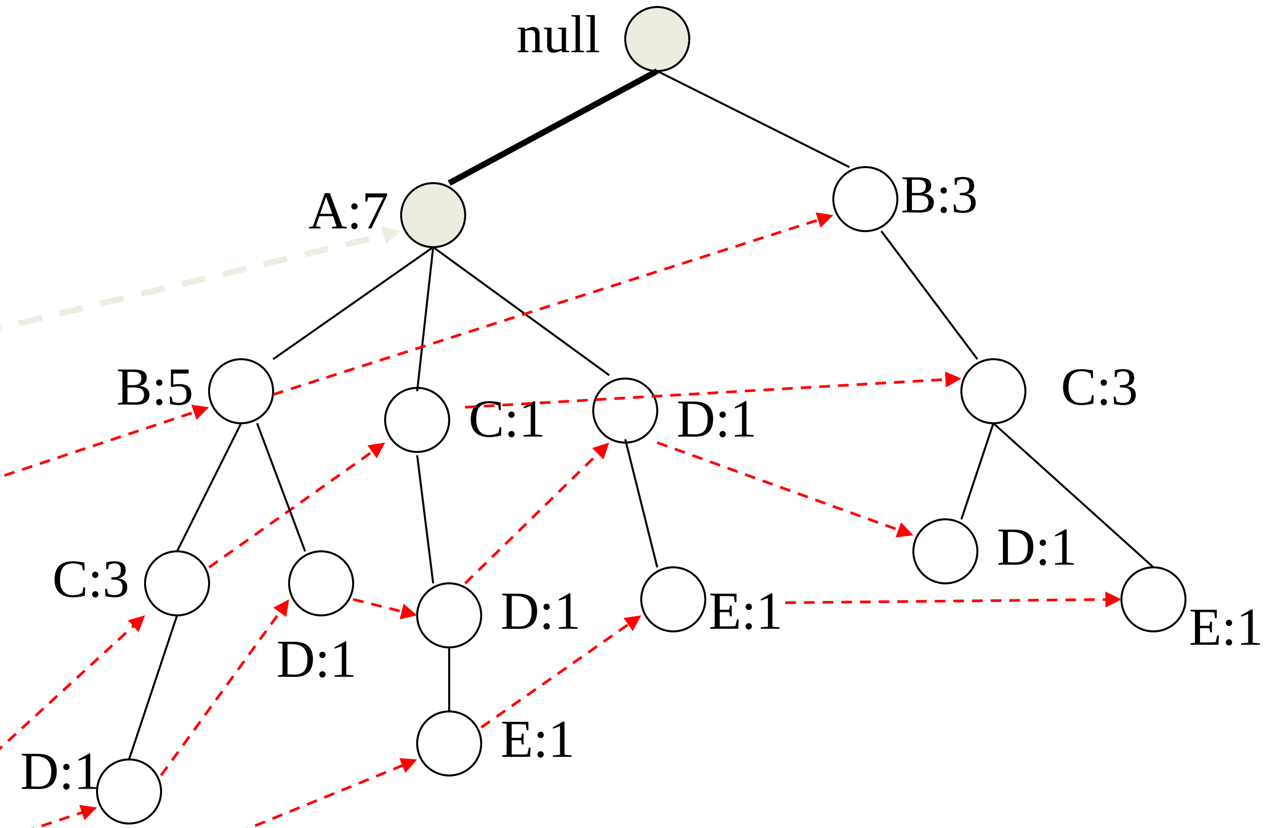
Header table

Item	Pointer
A	
B	
C	
D	
E	



Finding Frequent Itemsets

Ending in **A**



Header table

Item	Pointer
A	
B	
C	
D	
E	

Algorithm

- For each suffix X
- Phase 1
  - Construct the prefix tree for X as shown before, and compute the support using the header table and the pointers
- Phase 2
  - If X is frequent, construct the conditional FP-tree for X in the following steps
    1. Recompute support
    2. Prune infrequent items
    3. Prune leaves and recurse

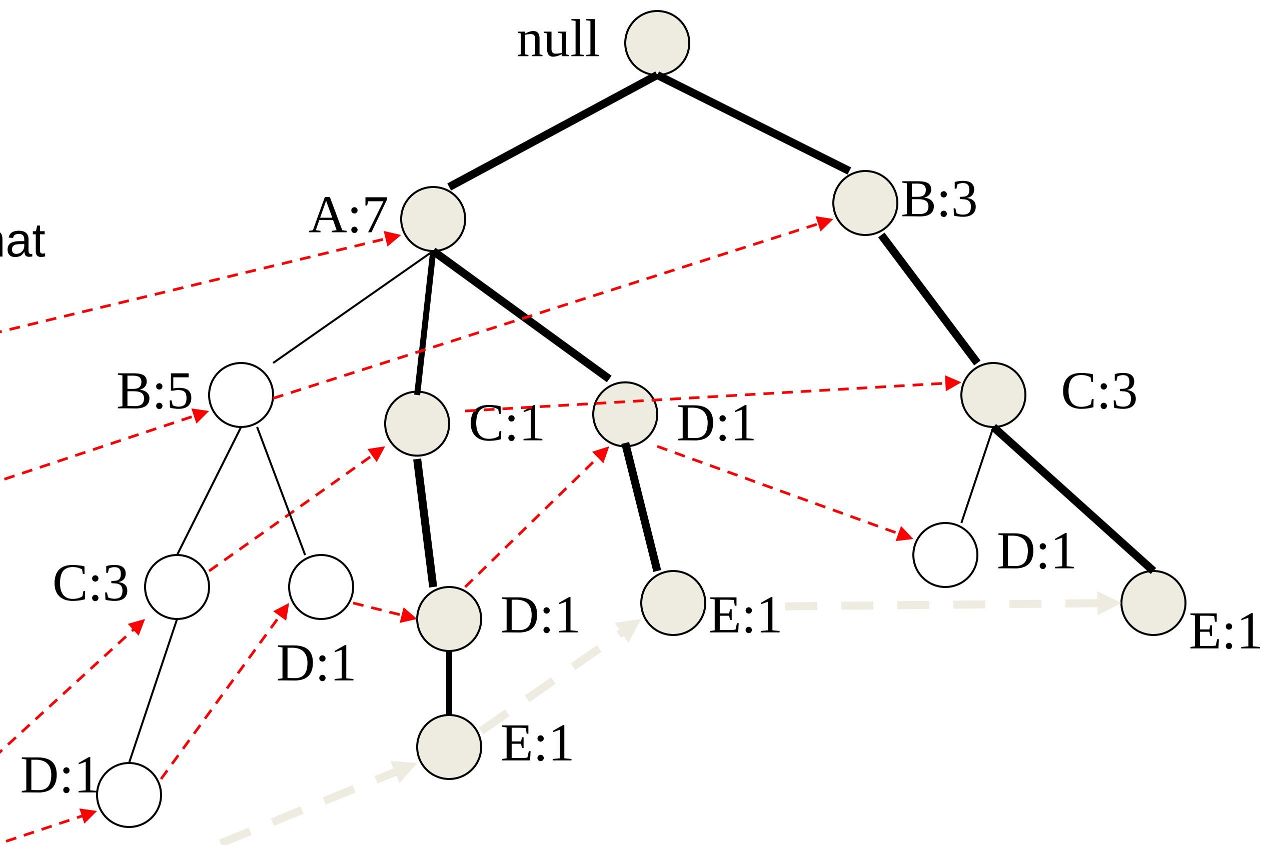
Example

Phase 1 – construct prefix tree

Find all prefix paths that contain E

Header table

Item	Pointer
A	
B	
C	
D	
E	



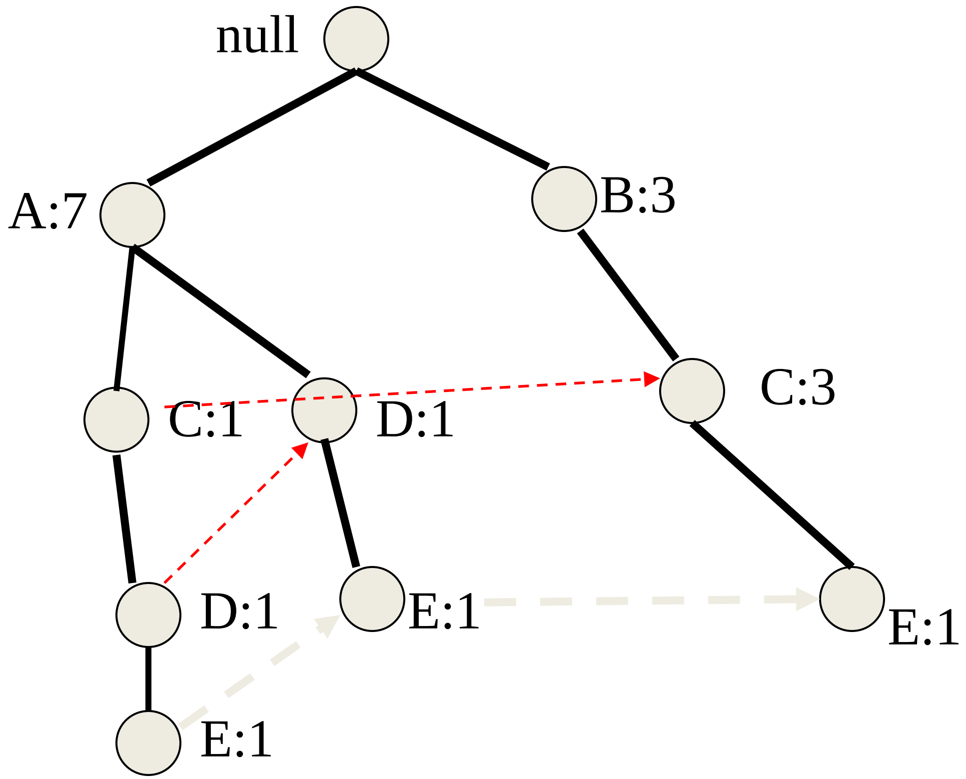
Suffix Paths for E:

{A,C,D,E}, {A,D,E}, {B,C,E}

Example

Phase 1 – construct prefix tree

Find all prefix paths that contain E



Prefix Paths for E:

{A,C,D,E}, {A,D,E}, {B,C,E}

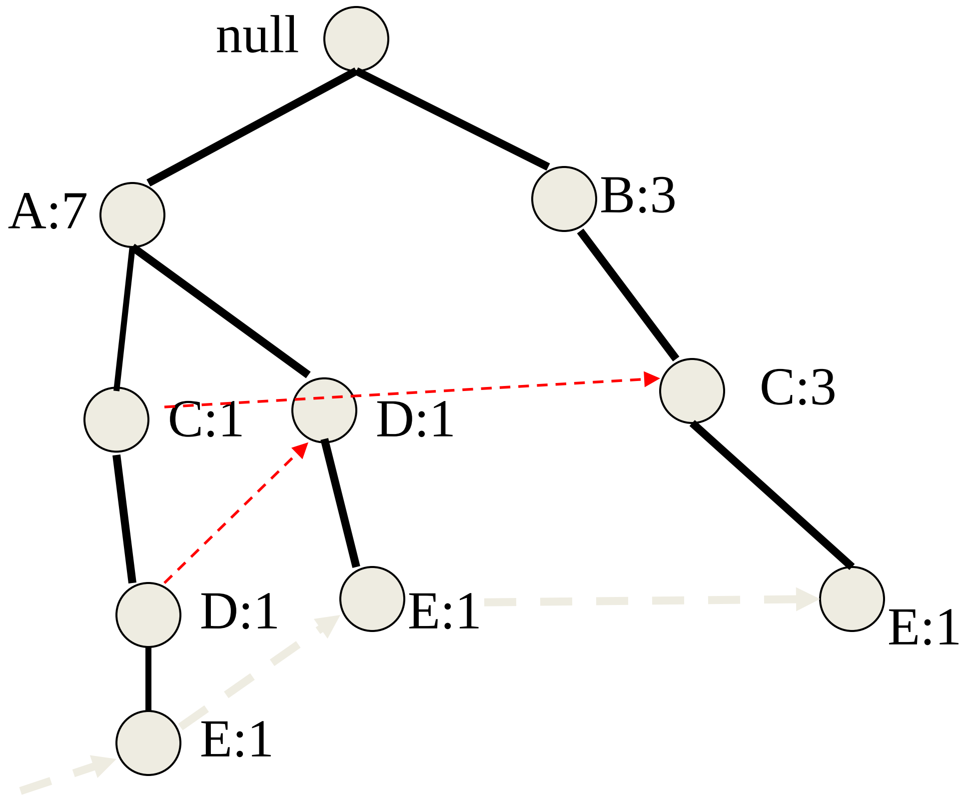
Example

Compute Support for E
(minsup = 2)

How?

Follow pointers while summing up counts: 1+1+1 = 3 > 2

E is frequent



{E} is frequent so we can now consider suffixes DE, CE, BE, AE

Example

E is frequent so we proceed with Phase 2

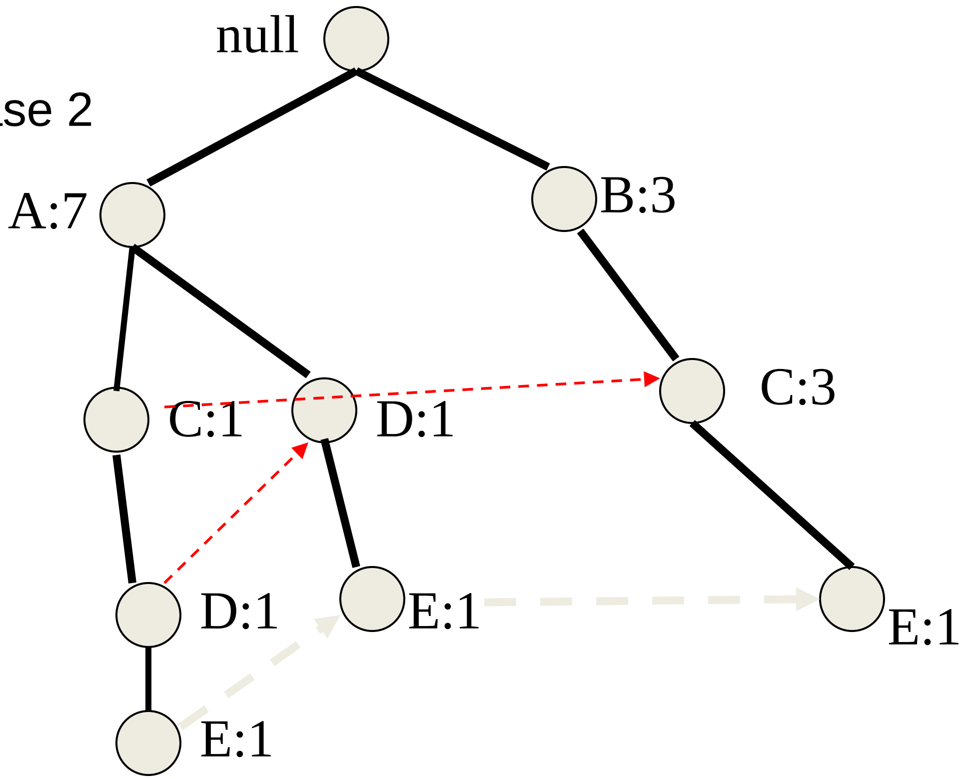
Phase 2

Convert the prefix tree of E into a conditional FP-tree

Two changes

(1) Recompute support

(2) Prune infrequent



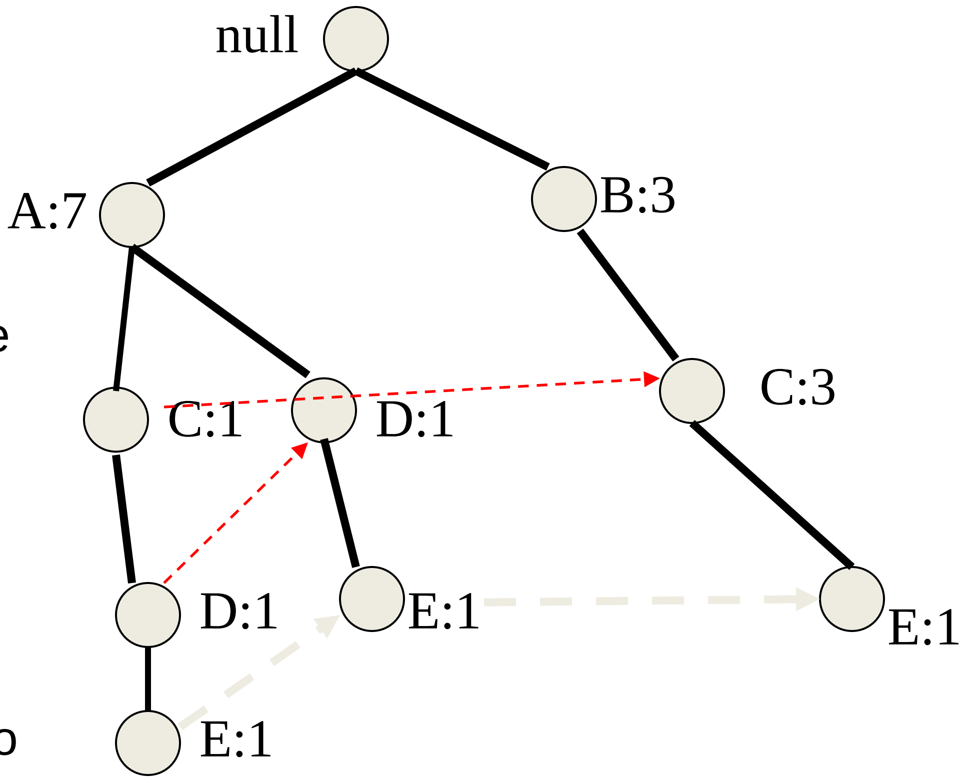
Example

Recompute Support

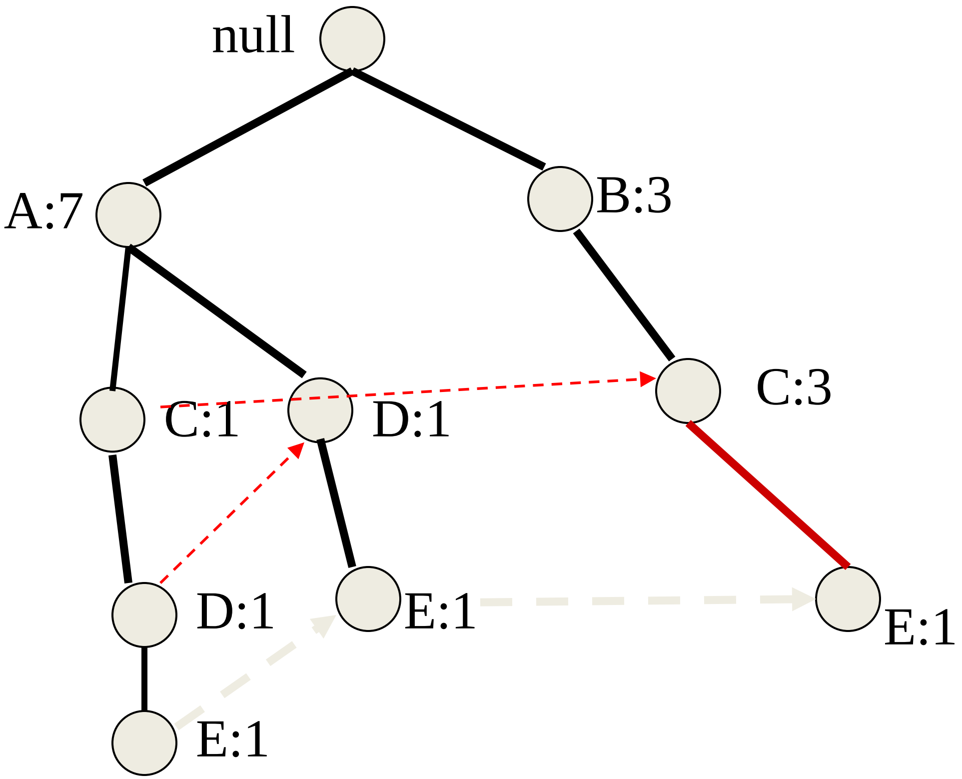
The support counts for some of the nodes include transactions that do not end in E

For example in null->B->C->E we count {B, C}

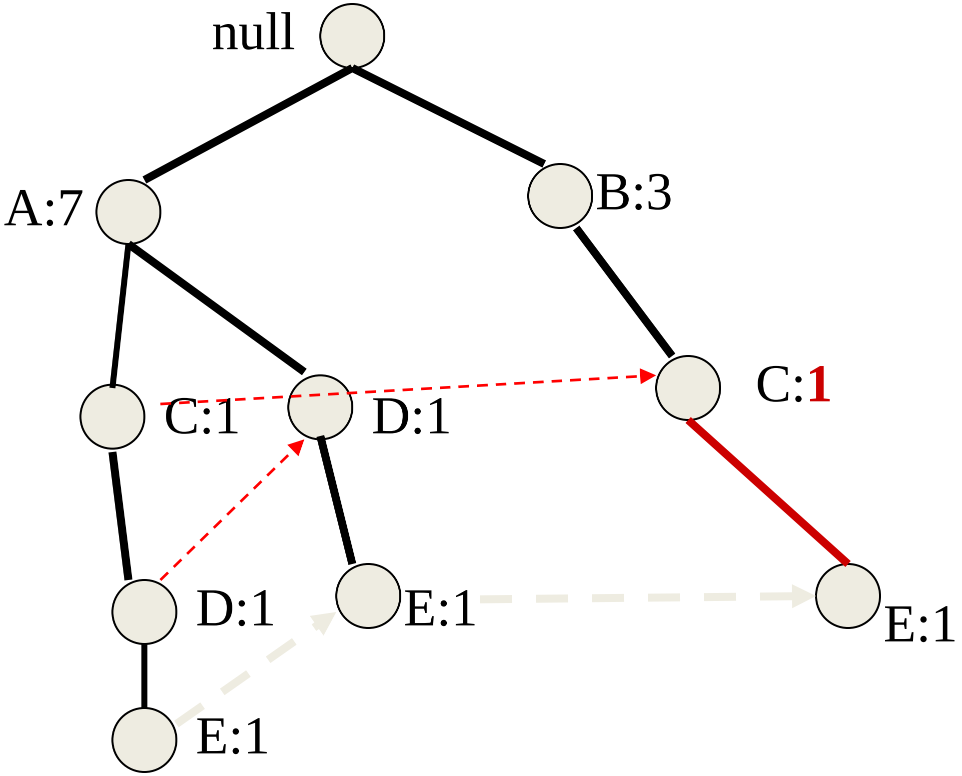
The support of any node is equal to the sum of the support of leaves with label E in its subtree



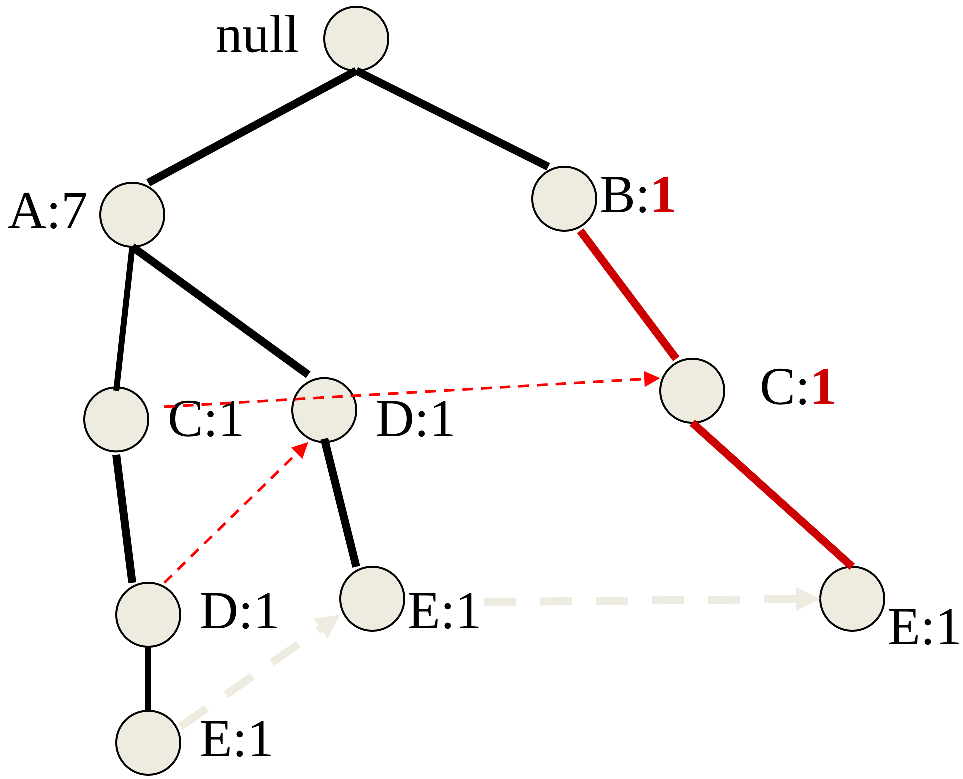
Example



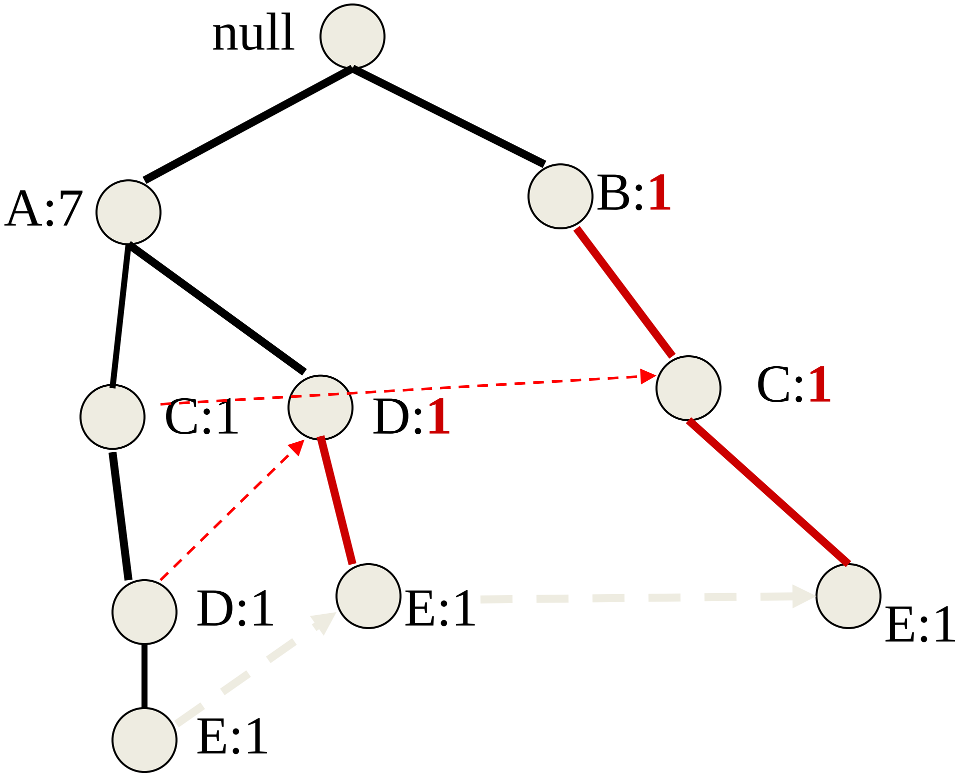
Example



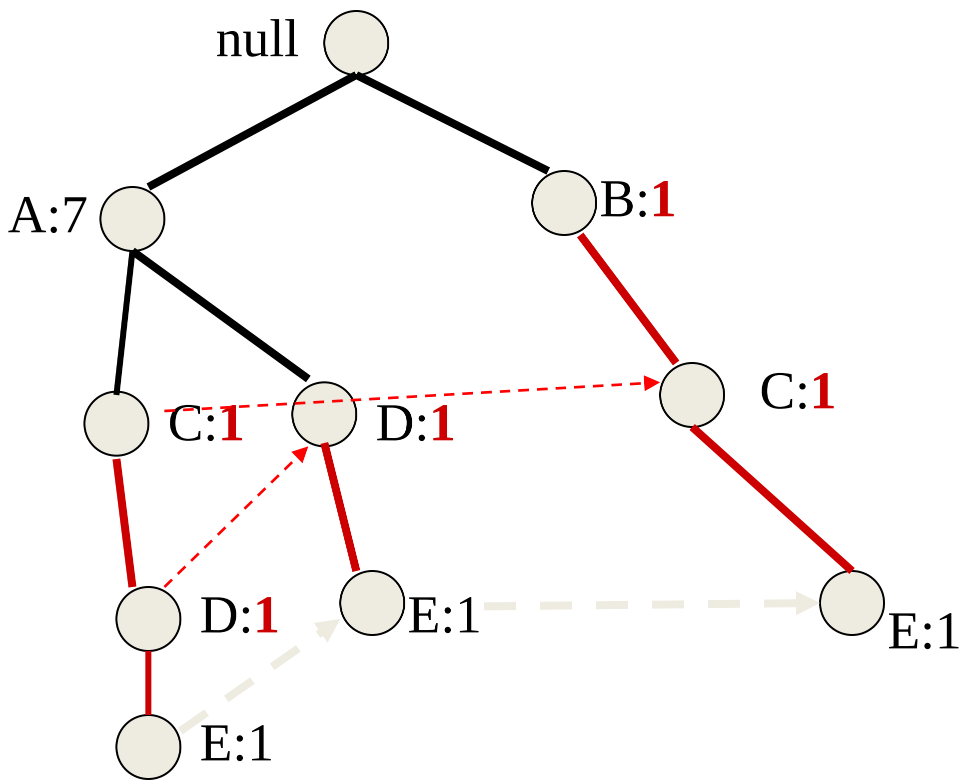
Example



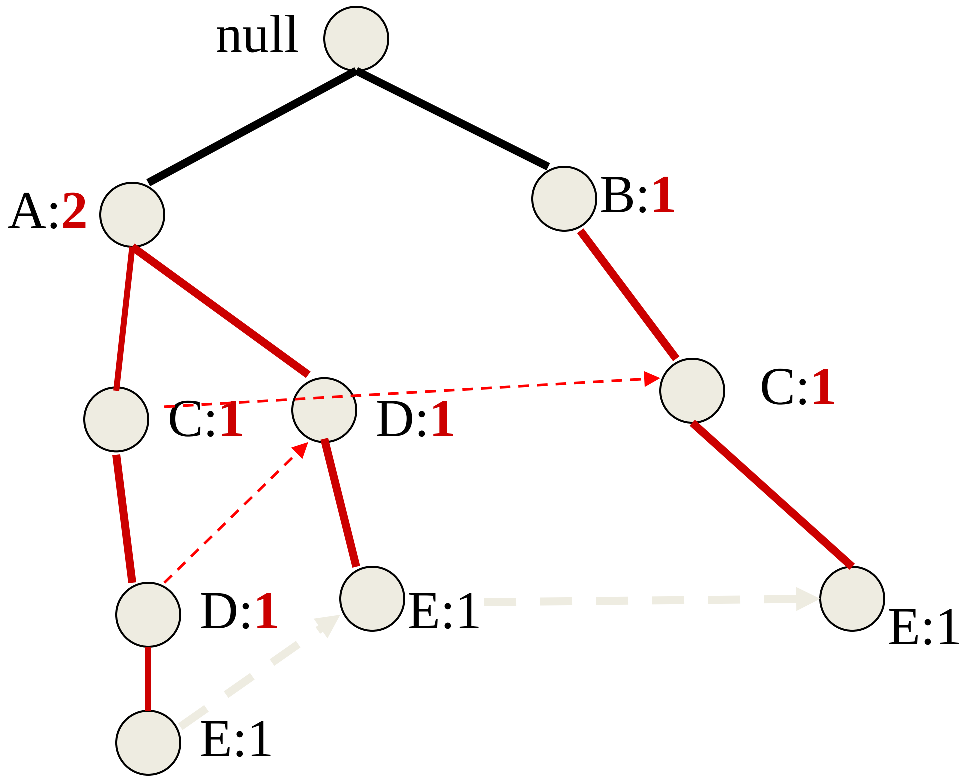
Example



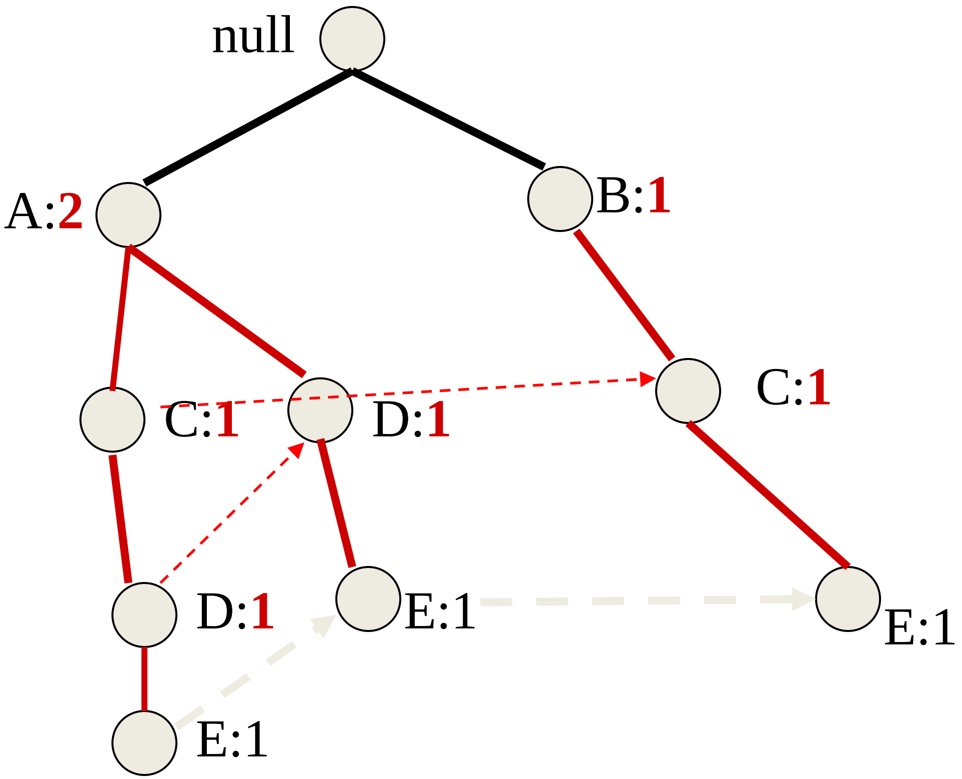
Example



Example



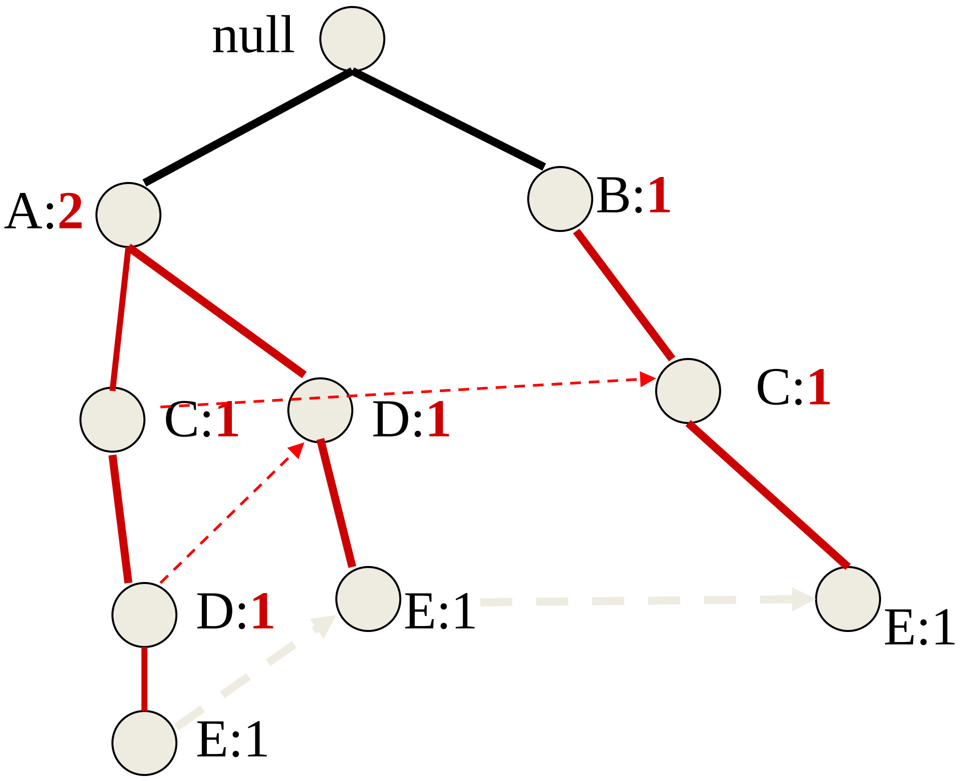
Example



Example

Truncate

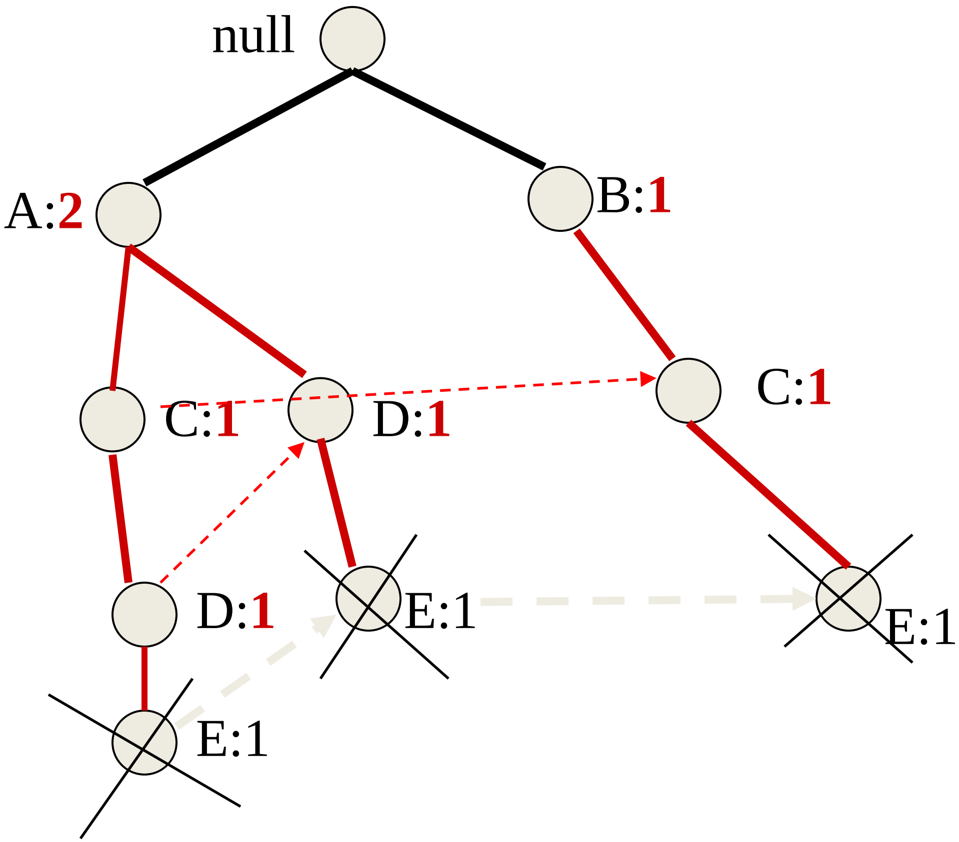
Delete the nodes of E



Example

Truncate

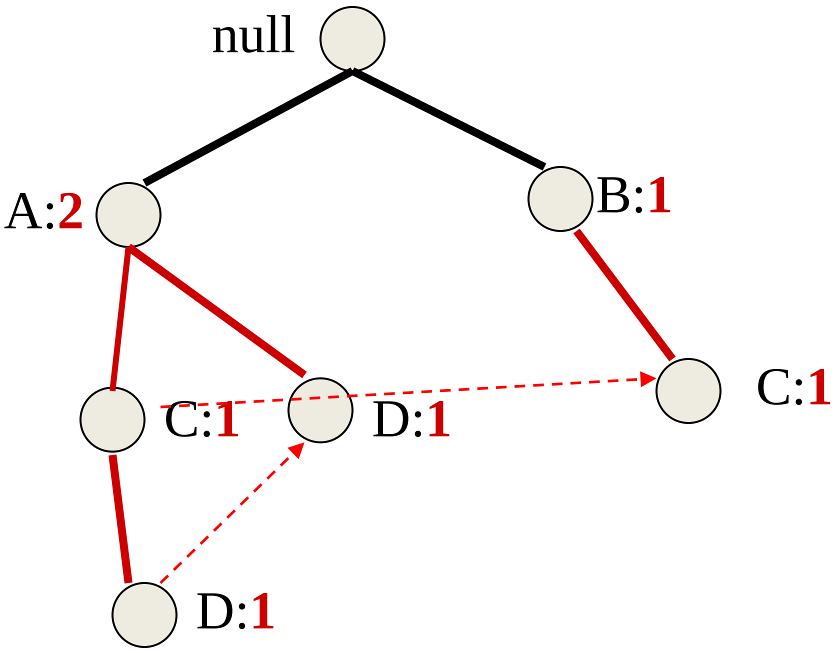
Delete the nodes of E



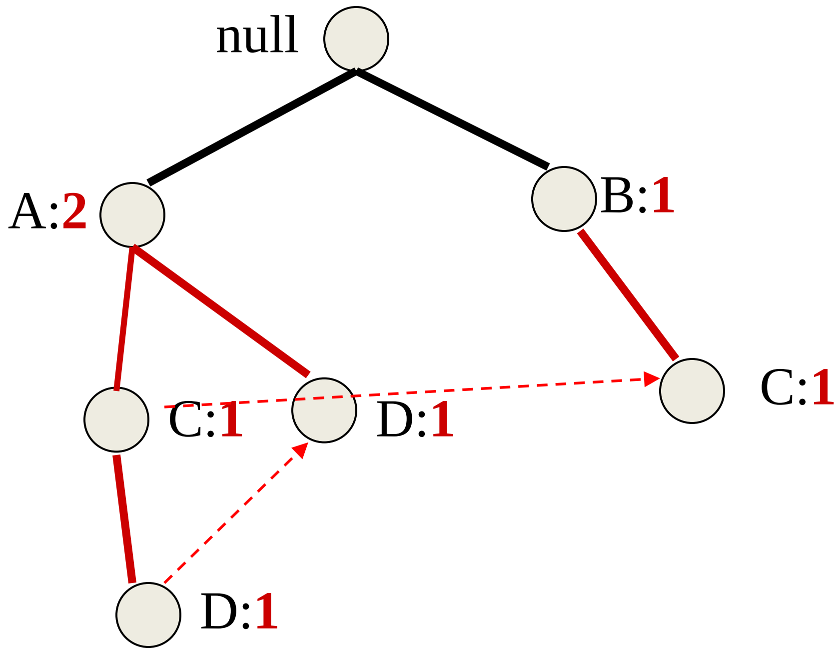
Example

Truncate

Delete the nodes of E



Example



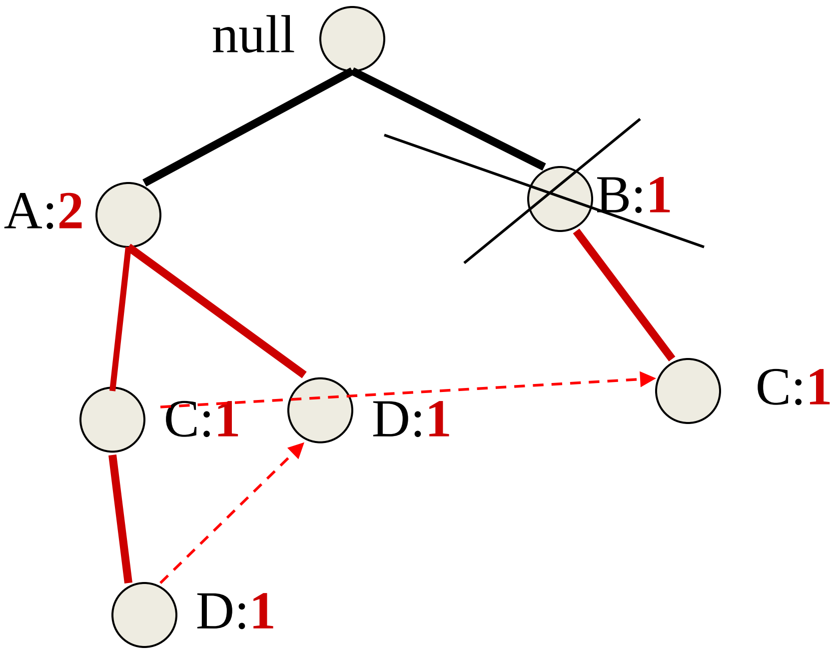
Prune infrequent

In the conditional FP-tree some nodes may have support less than minsup

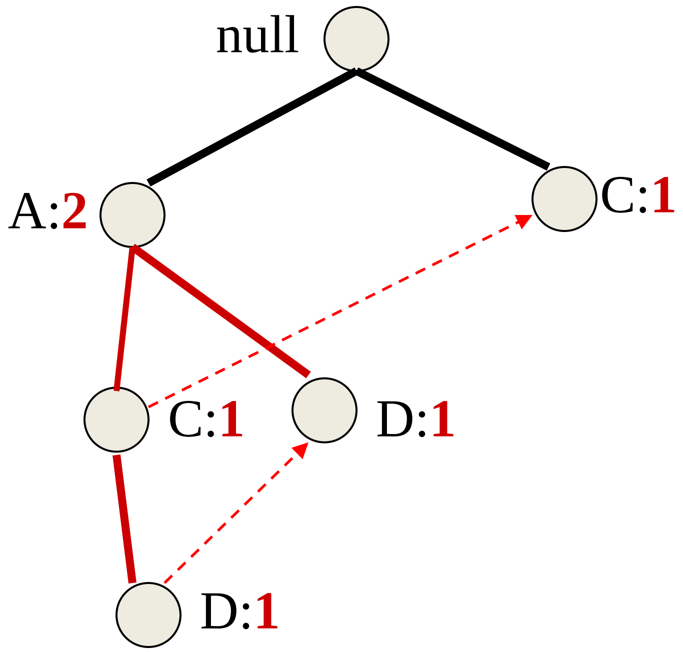
e.g., B needs to be pruned

This means that B appears with E less than minsup times

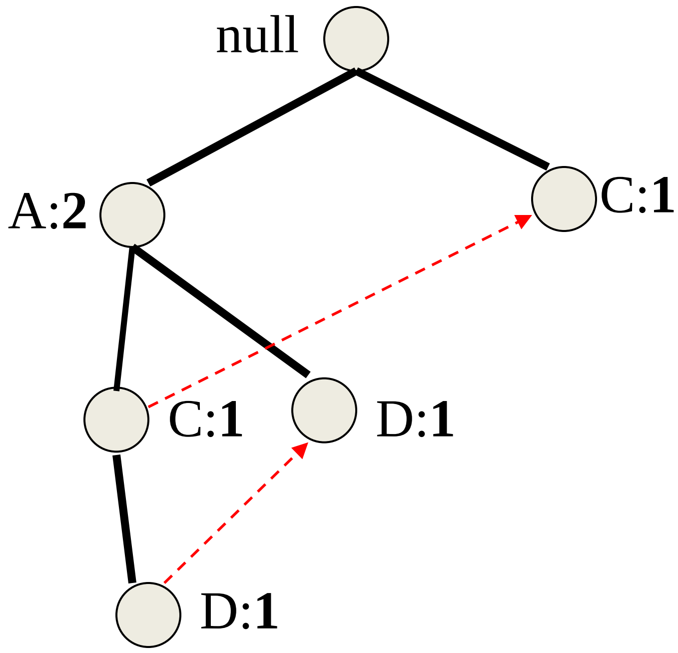
Example



Example



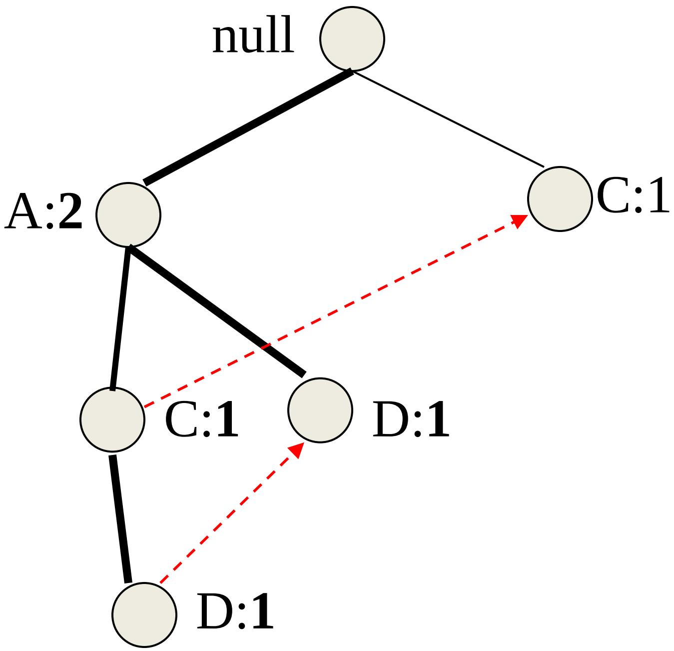
Example



The conditional FP-tree for E

Repeat the algorithm for {D, E}, {C, E}, {A, E}

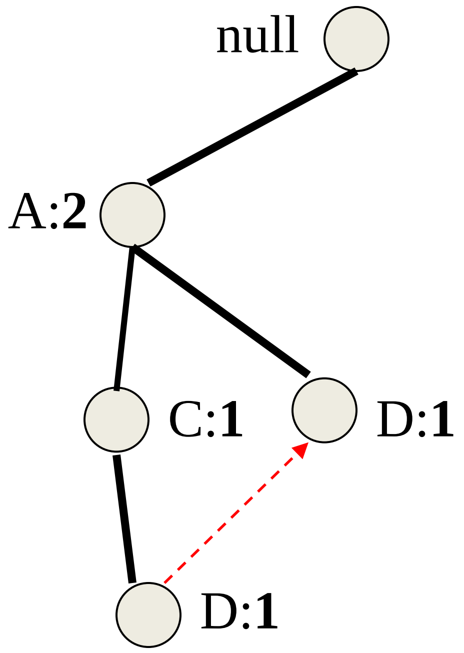
Example



Phase 1

Find all prefix paths that contain D (DE) in the conditional FP-tree

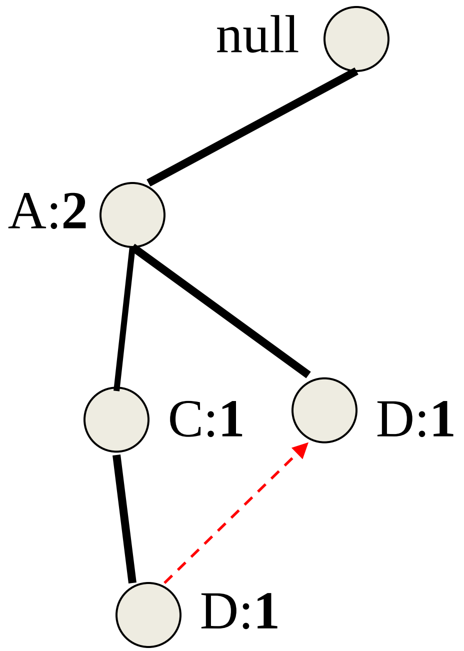
Example



Phase 1

Find all prefix paths that contain D (DE) in the conditional FP-tree

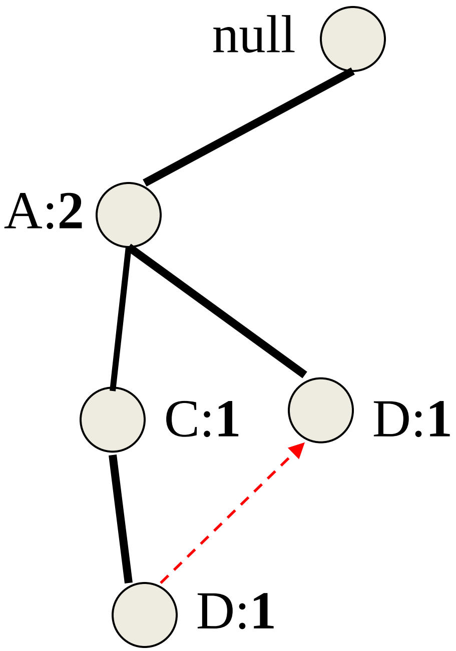
Example



Compute the support of {D,E} by following the pointers in the tree
1+1 = 2 ≥ 2 = minsup

{D,E} is frequent

Example



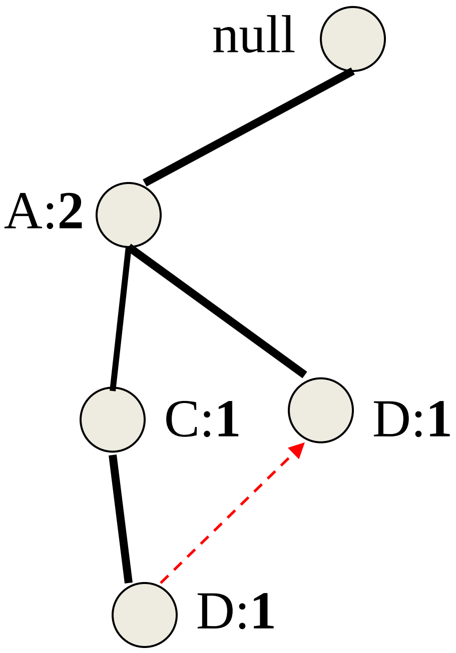
Phase 2

Construct the conditional FP-tree

1. Recompute Support
2. Prune nodes

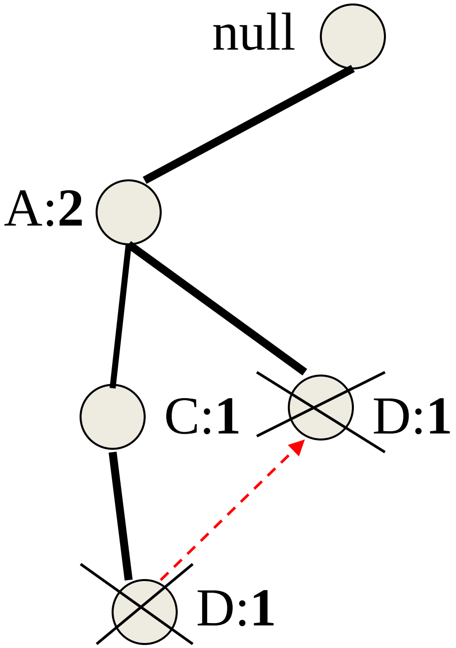
Example

Recompute support

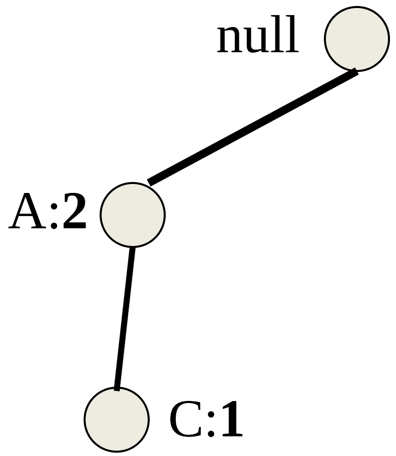


Example

Prune nodes

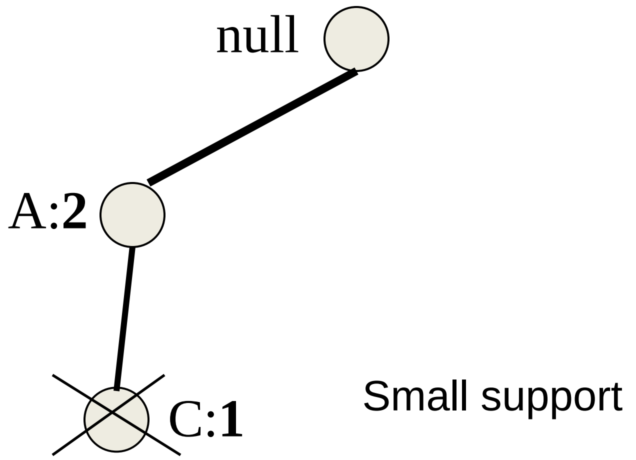


Example



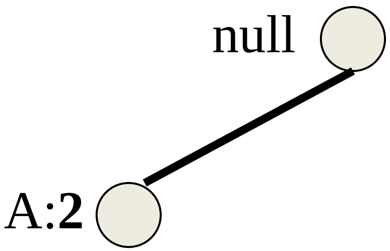
Prune nodes

Example



Prune nodes

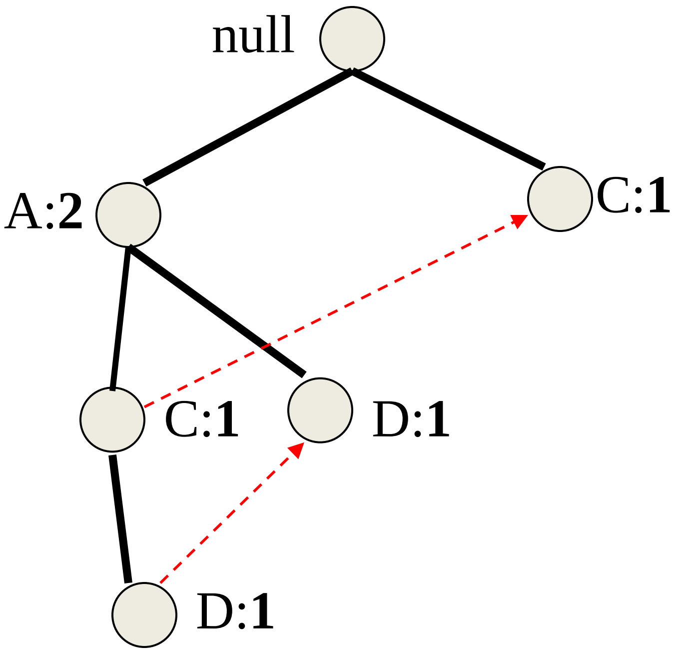
Example



Final condition FP-tree for {D,E}

The support of A is ≥ minsup so {A,D,E} is frequent
Since the tree has a single node we return to the next subproblem

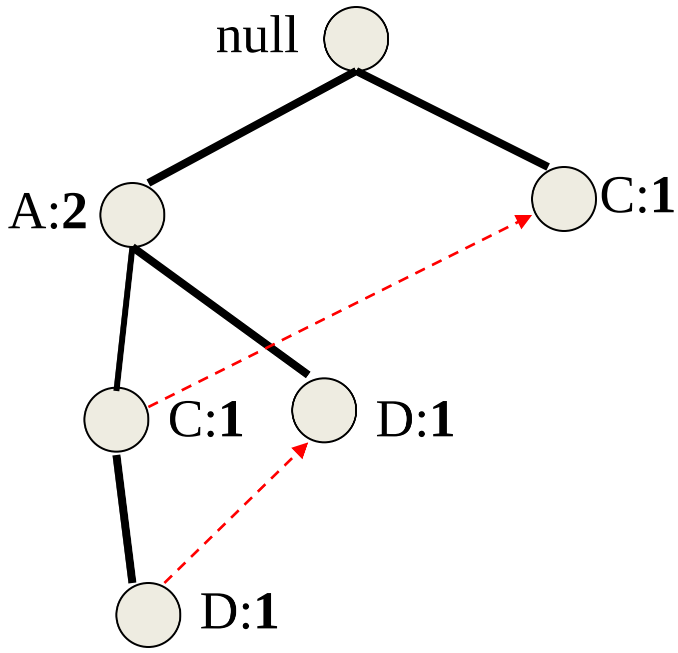
Example



The conditional FP-tree for E

We repeat the algorithm for ~~{D,E}~~, {C,E}, {A,E}

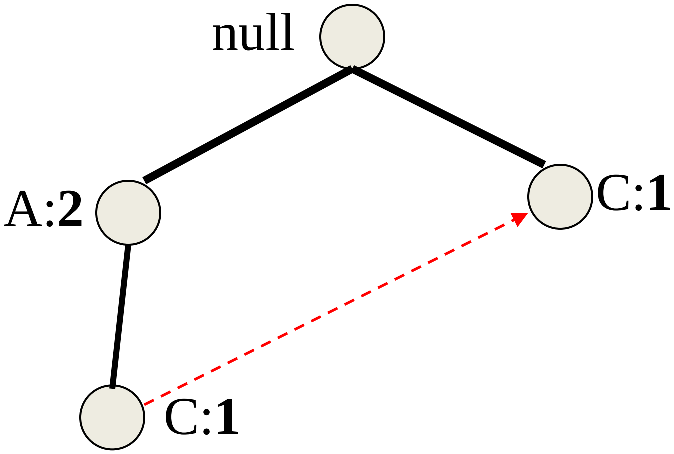
Example



Phase 1

Find all prefix paths that contain C (CE) in the conditional FP-tree

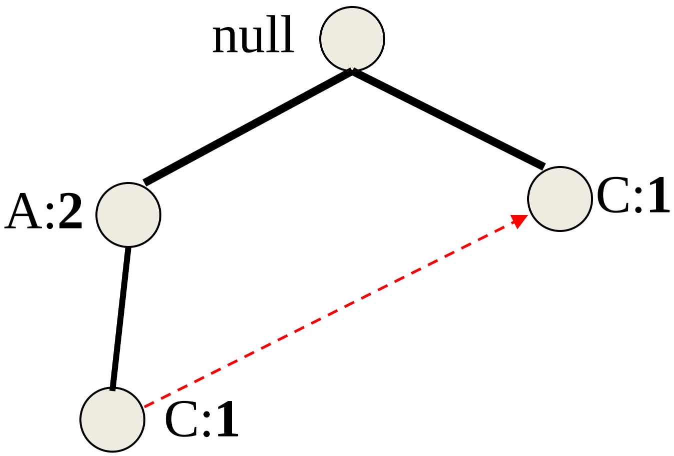
Example



Phase 1

Find all prefix paths that contain C (CE) in the conditional FP-tree

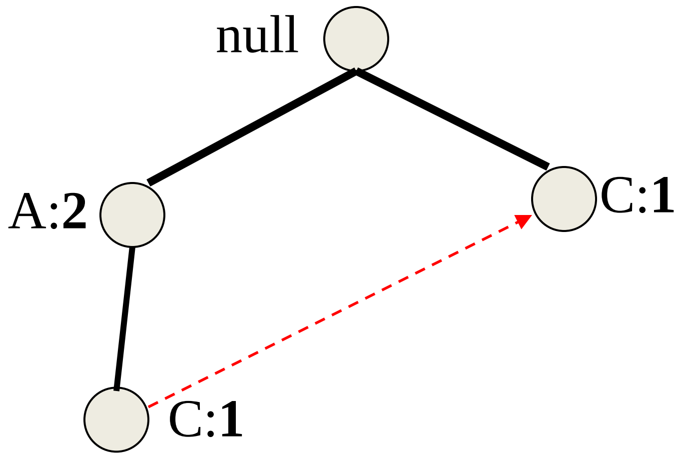
Example



Compute the support of {C,E} by following the pointers in the tree
1+1 = 2 ≥ 2 = minsup

{C,E} is frequent

Example

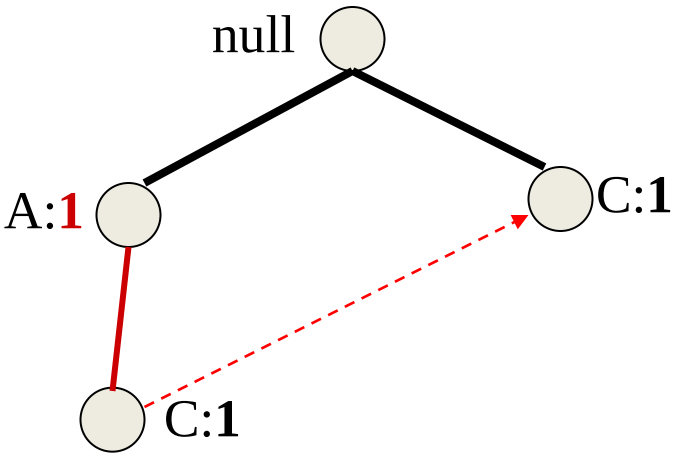


Phase 2

Construct the conditional FP-tree

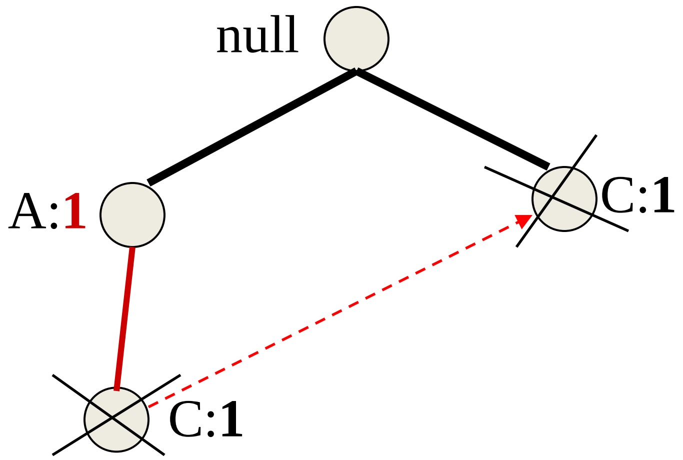
1. Recompute Support
2. Prune nodes

Example



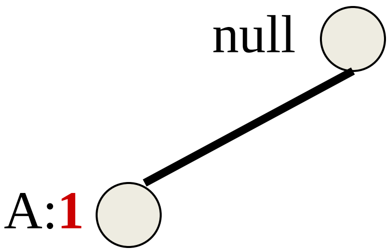
Recompute support

Example



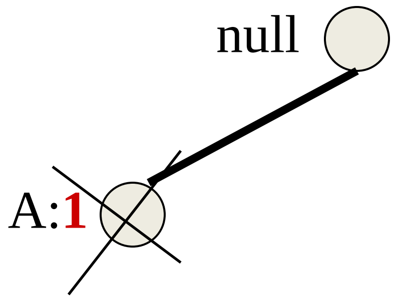
Prune nodes

Example



Prune nodes

Example



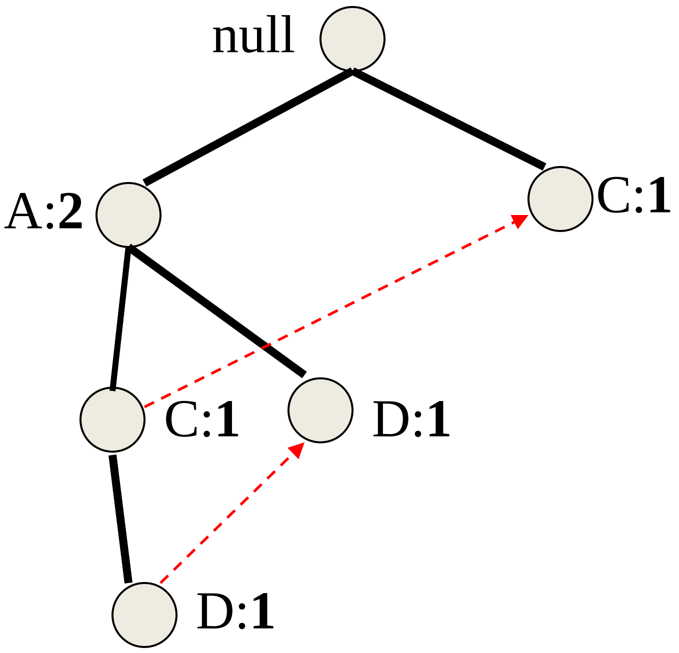
Prune nodes

Example

Prune nodes

Return to the previous subproblem

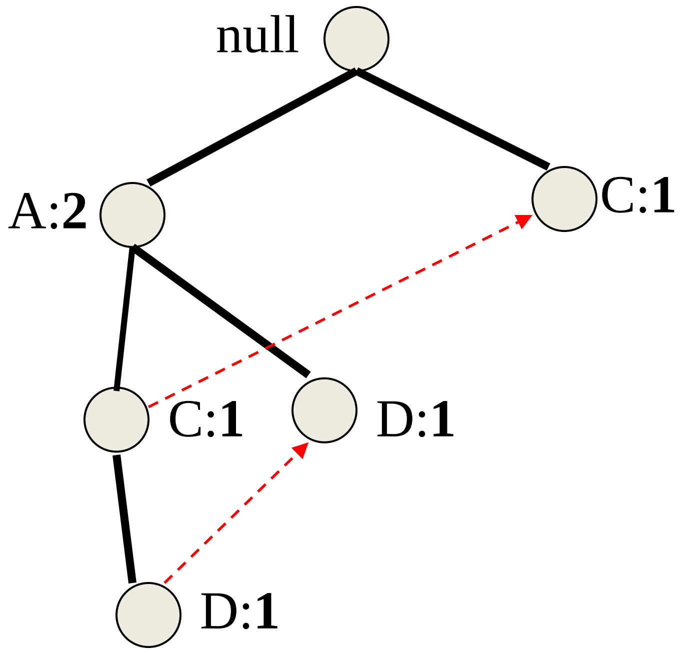
Example



The conditional FP-tree for E

We repeat the algorithm for ~~{D,E}, {C,E},~~ {A,E}

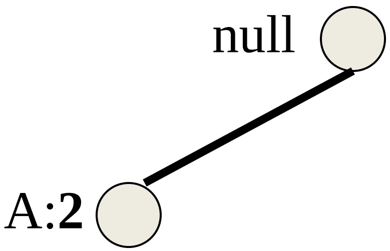
Example



Phase 1

Find all prefix paths that contain A (AE) in the conditional FP-tree

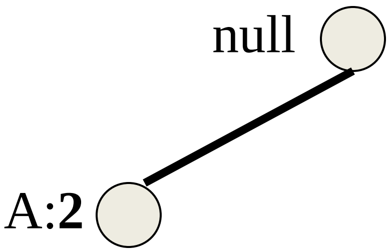
Example



Phase 1

Find all prefix paths that contain A (AE) in the conditional FP-tree

Example



Compute the support of {A,E} by following the pointers in the tree
2 ≥ minsup

{A,E} is frequent

There is no conditional FP-tree for {A,E}

Example

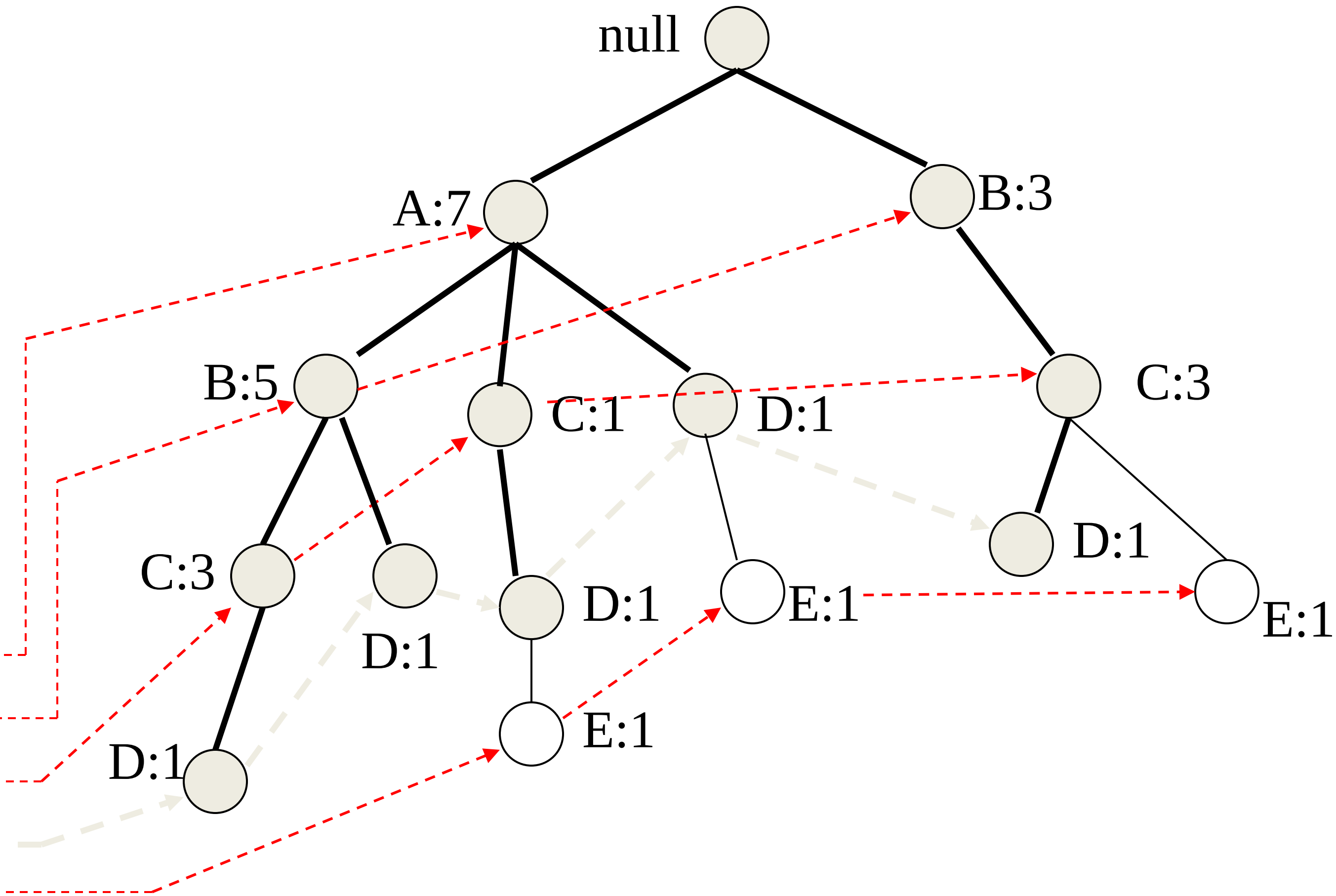
- So for E we have the following frequent itemsets

{E}, {D,E}, {A,D,E} , {C,E}, {A,E}

- We proceed with D

Example

Ending in **D**



Header table

Item	Pointer
A	
B	
C	
D	
E	

Example

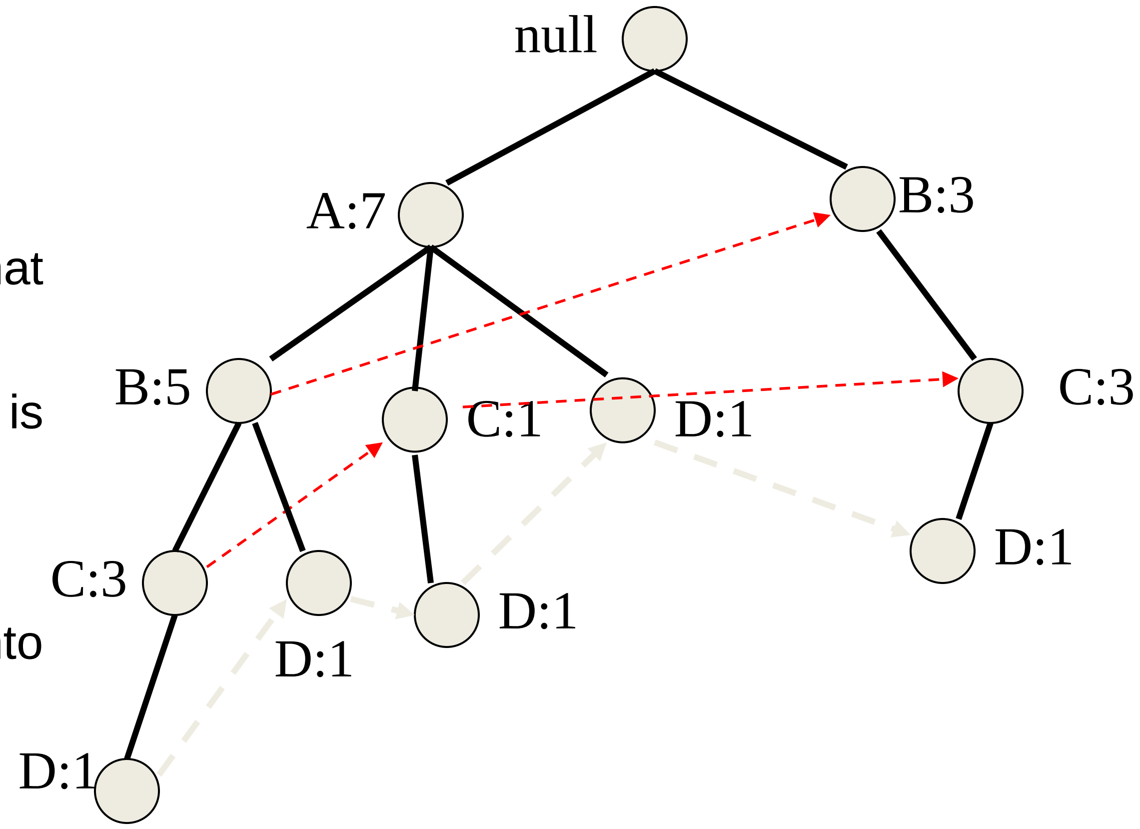
Phase 1 – construct prefix tree

Find all prefix paths that contain D

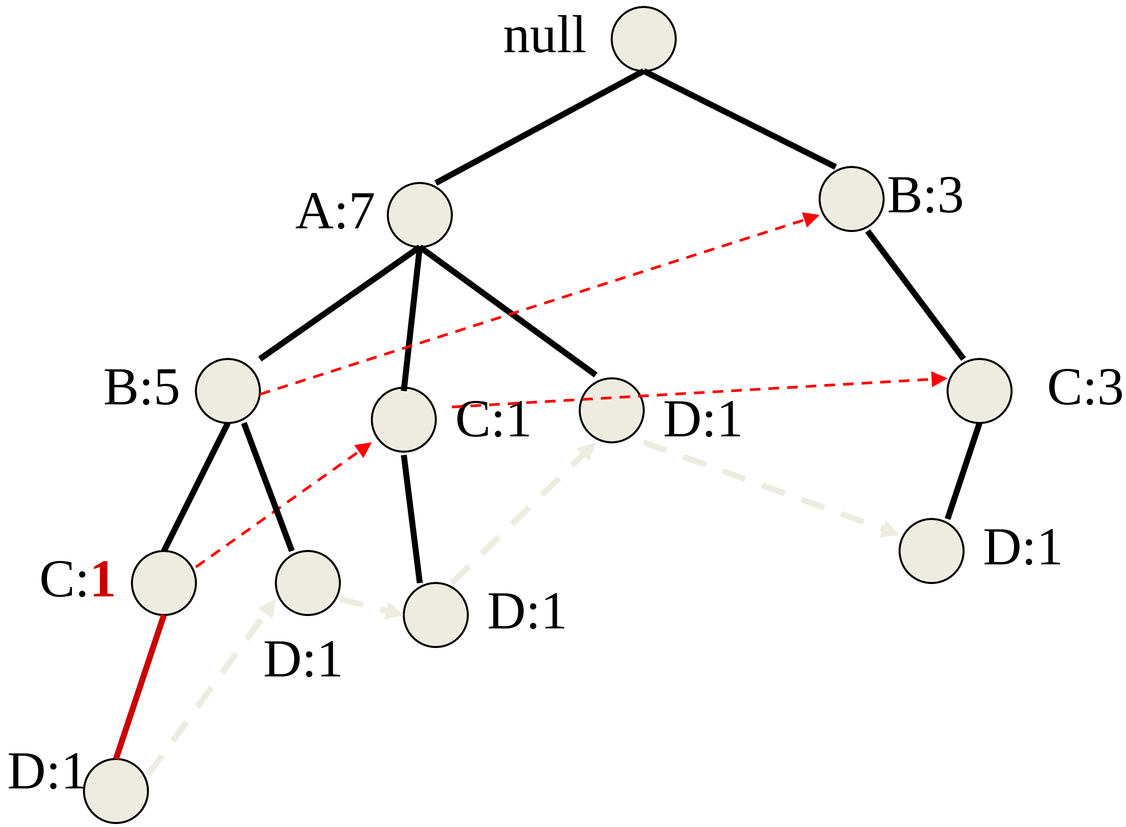
Support 5 > minsup, D is frequent

Phase 2

Convert prefix tree into conditional FP-tree

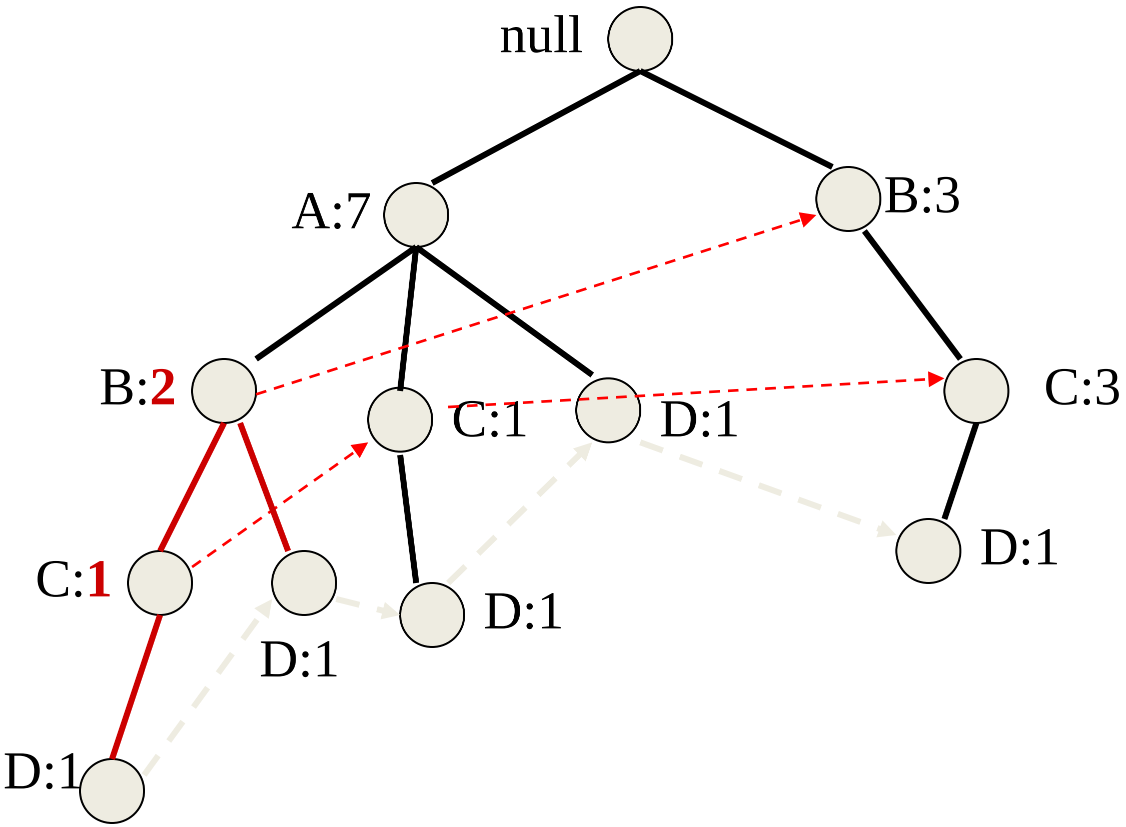


Example



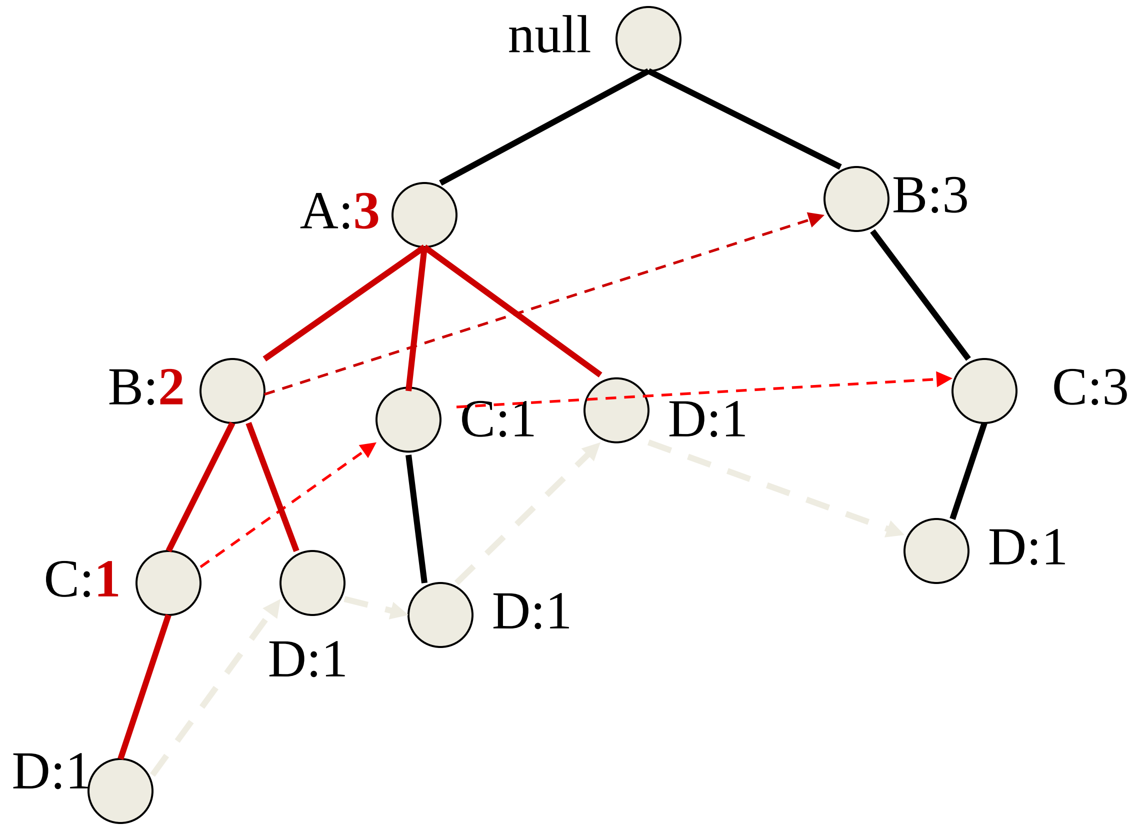
Recompute support

Example



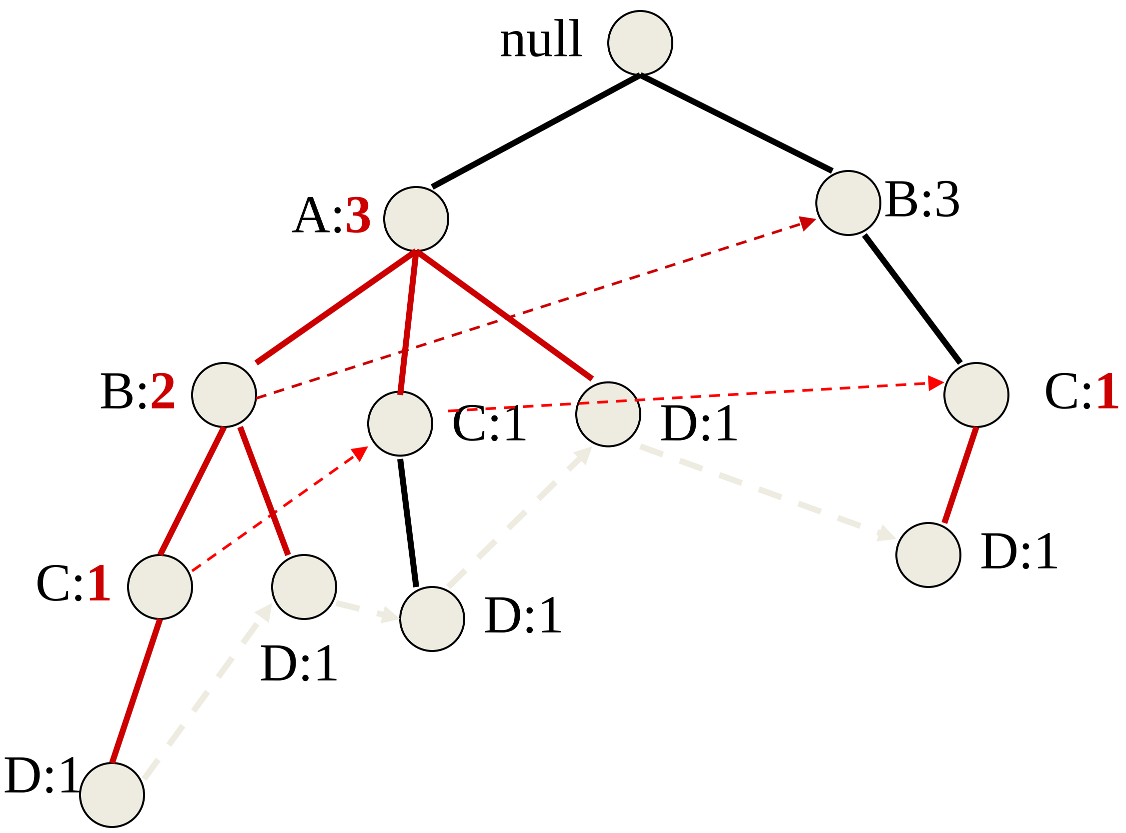
Recompute support

Example



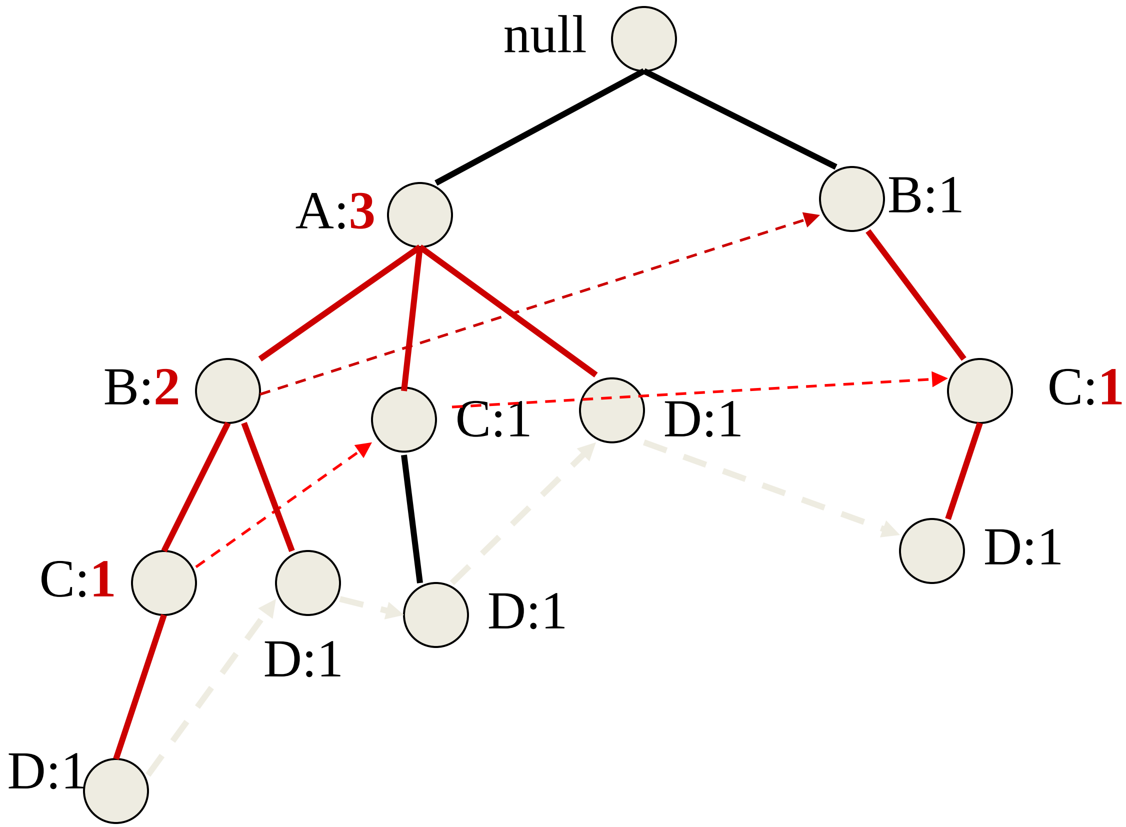
Recompute support

Example



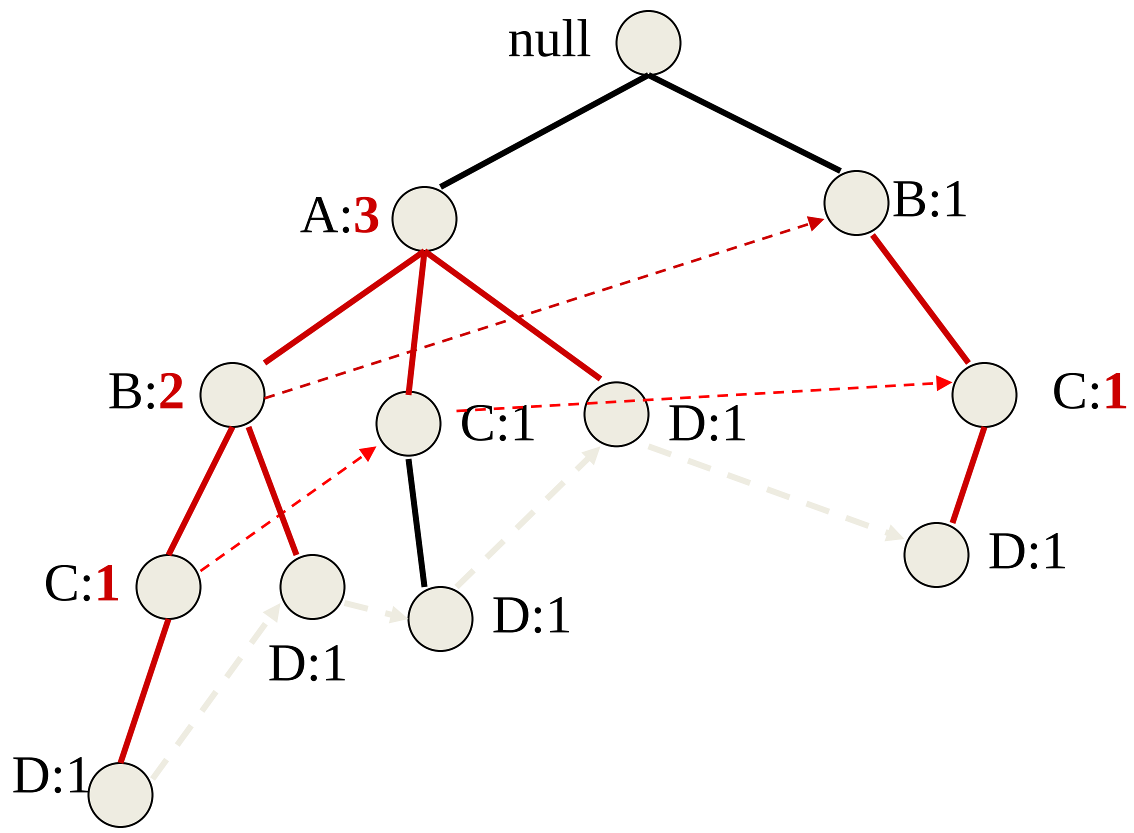
Recompute support

Example



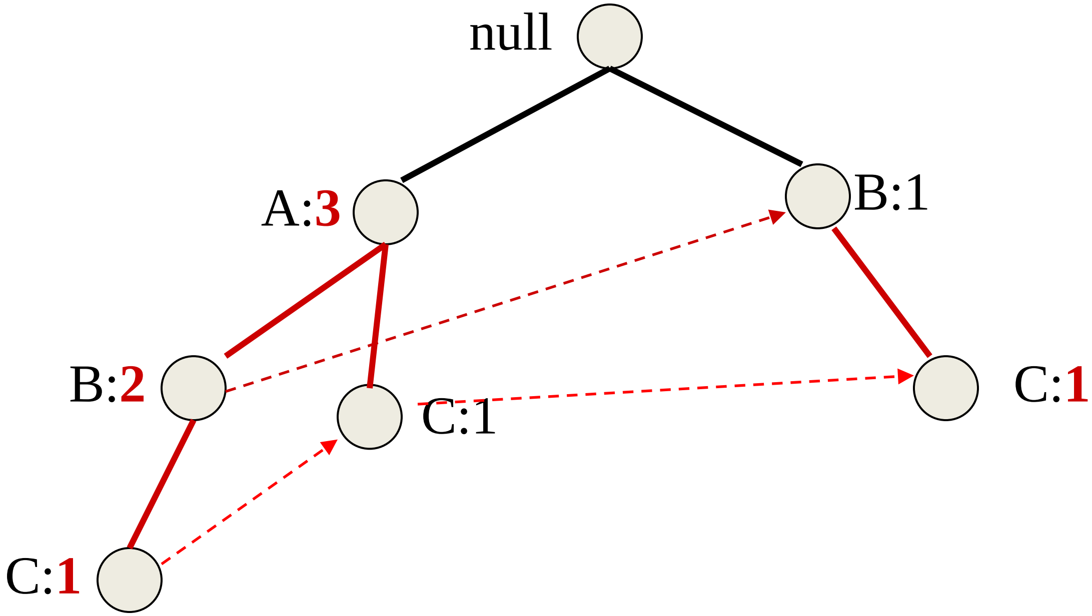
Recompute support

Example



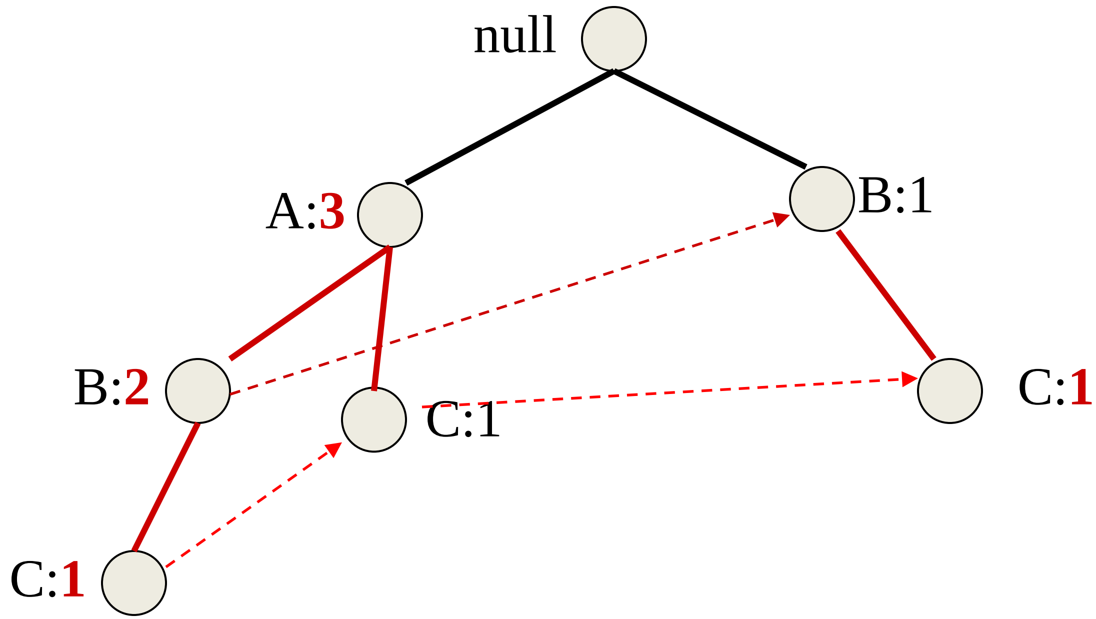
Prune nodes

Example



Prune nodes

Example



Construct conditional FP-trees for {C,D}, {B,D}, {A,D}

And so on….

Observations

- At each recursive step we solve a subproblem
  - Construct the prefix tree
  - Compute the new support
  - Prune nodes
- Subproblems are disjoint so we never consider the same itemset twice
- Support computation is efficient – happens together with the computation of the frequent itemsets.

Observations

- The efficiency of the algorithm depends on the compaction factor of the dataset

- If the tree is bushy then the algorithm does not work well, it increases a lot of number of subproblems that need to be solved.

FREQUENT ITEMSET RESEARCH

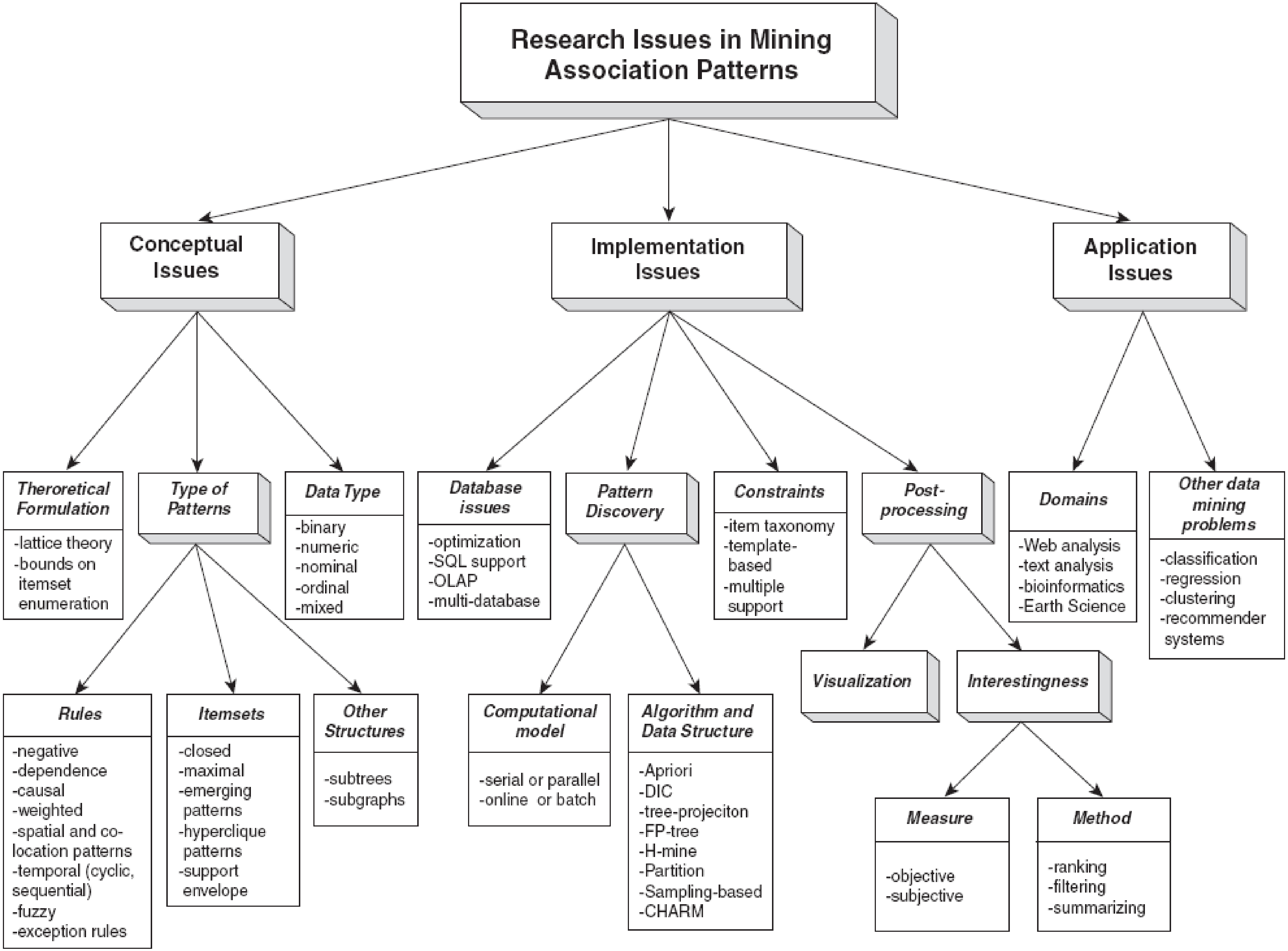


Figure 6.31. A summary of the various research activities in association analysis.